REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	March 1993	Master's Thesis

4. TITLE AND SUBTITLE
NETWORKED WORKSTATIONS AND PARALLEL PROCESSING UTILIZING FUNCTIONAL LANGUAGES

5. FUNDING NUMBERS

6. AUTHOR(S)
FOX, Stanley L. II

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Naval Postgraduate School
Monterey, CA 93943-5000

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES
The views expressed in this thesis are those of the author and no do reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)
Alternative computer architectures are necessary to replace the traditional 'von Neumann' computer organization in order to obtain large increases in performance. The traditional 'von Neumann' architecture uses a timer based (e.g., the program counter), sequentially programmed, single processor approach to problem solving. Today's new hardware technology allows for the utilization of multiple processors. By programming and operating these processors in parallel, this alternative architecture will provide for greater computing speed, improved system reliability, enhanced software manageability, and a more cost-effective approach than our present computing practices.

14. SUBJECT TERMS	15. NUMBER OF PAGES
parallel processing; distributed computing; functional languages; parallel programming languages	124
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLAS	UNCLAS	UNCLAS	UL

Approved for public release, distribution is unlimited

Networked Workstations and Parallel Processing
Utilizing Functional Languages

by

Stanley L. Fox II
Lieutenant, United States Navy
B.S., University of Oklahoma, 1986

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL
March 1993

ABSTRACT

Alternative computer architectures are necessary to replace the traditional 'von Neumann' computer organization in order to obtain large increases in performance. The traditional 'von Neumann' architecture uses a timer based (e.g., the program counter), sequentially programmed, single processor approach to problem solving. Today's new hardware technology allows for the utilization of multiple processors. By programming and operating these processors in parallel, this alternative architecture will provide for greater computing speed, improved system reliability, enhanced software manageability, and a more cost-effective approach than our present computing practices.

TABLE OF CONTENTS

ACKNOWLEDGEMENT

I would like to extend my deepest appreciation to Professor Fouts for his guidance and insight throughout this thesis research. I also extend thanks to the ECE Computer Center, specifically Brad Polk for his assistance and support with the SUN network. I would also like to thank the ECE Department faculty and staff. Additionally, I would like to extend my deepest gratitude to Ms. Karen Callaghan for her staunch support and guidance throughout my entire tour at the Naval Postgraduate School (Karen, you kept me out of a lot of trouble).

I. INTRODUCTION

A. PROBLEM STATEMENT

Alternative computer architectures are necessary to replace the traditional 'von Neumann' computer organization in order to obtain large increases in performance. The traditional 'von Neumann' architecture uses a timer based (e.g., the program counter), sequentially programmed, single processor approach to problem solving. Today's new hardware technology allows for the utilization of multiple processors. By programming and operating these processors in parallel, this alternative architecture will provide for greater computing speed, improved system reliability, enhanced software manageability, and a more cost-effective approach than our present computing practices. [Ref. 1]

Many of the advances towards parallel processing have been hardware related, and the software to properly exploit a parallel architecture is still in the development stages. Digital Equipment Corporation, the University of Manchester (England), Lawrence Livermore National Laboratory, and Colorado State University worked cooperatively beginning in 1983 to develop a programming language for parallel numerical computation. [Ref. 2] The outcome of this project was the programming language SISAL (Streams and Iterations in a Single Assignment Language). The SISAL language is a functionally oriented programming language. The primary goal of SISAL is to achieve sequential and

parallel execution performance, depending upon the hardware architecture, superior to programs written in conventional languages. [Ref. 2]

B. OBJECTIVE

The goal of this thesis is to determine the possibility of exploiting the multiprocessor architecture found on a network of workstations by using the SISAL Language. The reader will obtain a better understanding of parallel processing by the basic review of the dataflow approach to parallel processing provided in Chapter II. This review will introduce the use of the dataflow graph for ease in portraying parallel processes.

This discussion will then be followed by an introduction to functional languages in Chapter III. The use of functional languages is receiving wide attention in the area of scientific numerical processing. One of the most prevalent benefits of the use of functional languages is that they free the programmer from writing explicitly parallel code. A functional language is capable of exploiting the inherent parallelism of an algorithm without the necessity for the programmer to specify where the parallelism is to occur.

As the goal of this thesis was to determine if SISAL, a functional language, could be utilized on a distributed system, specifically a SUN Network, the discussion would not be complete without introducing the ECE Department's SUN Network. Chapter IV discusses loosely coupled distributed systems in general, and the SUN Network in particular.

The SISAL Language is introduced in Chapter V. An introduction to the Optimizing SISAL Compiler (OSC) and how it operates leads into a discussion of the research. The ultimate answer to the research lies in the application of the OSC to the distributed Network of SUN Workstations. This is followed by the conclusions formulated by the author in Chapter VI, as well as some insight towards where this study could be further developed.

II. PARALLEL COMPUTING AND THE DATAFLOW APPROACH

A. PARALLELISM IN SOFTWARE

Within any software developed, there are two types of parallelism possible. The first, or regular parallelism, is that which occurs when a common set of operations, either simple or complex, is applied to many separate sets of data. An example of this type of parallelism can be found in the execution of WHILE-DO Loops, or FOR Loops. The second type of parallelism, known as irregular parallelism, is found when different operations, again simple or complex, are applied to either common or separate sets of data. This type of parallelism is demonstrated in the following block of assignment statements:

```
A := E - G;
B := H * Z;
C := E * H + F;
D := E + G;
```

Hardware schemes that exploit the first type of parallelism are known as single-instruction-stream, multiple-data-stream (SIMD) systems, and those that exploit the second type of parallelism are called multiple-instruction-stream, multiple-data-stream (MIMD) systems. [Ref. 1]

The construction of parallel computers that exploit regular parallelism is fairly commonplace. However, it has proven to be surprisingly difficult to find software, or programs, that will provide sufficient parallelism of the desired nature on a continuous basis. Hence, it has been found that applications execute with

varying degrees of speed with regular parallel expressions being executed more rapidly, and the other sections being executed more slowly. Since the slower sections tend to dominate the overall performance improvement, for the most part, the improvement is only a small fraction of that originally intended.

Although few systems have been developed to exploit irregular parallelism, those mechanisms that do achieve this task are also capable of handling regular parallelism. It is in this area though, that much research is being conducted. Parallelism at the process level, and implemented on shared-memory, or message passing processors have been developed, and use programming languages like Concurrent Pascal, Modula, etc. [Ref. 1] Dataflow systems tend to exploit irregular parallelism at a lower level, a level which approximates the conventional machine code level.

B. DATAFLOW NOTATION, AND PROGRAMS AS GRAPHS

Regardless of the nature of parallelism to be exploited, it is of key importance to provide an effective notation to express the potential for parallelism programs in any system developed. The following development of notation for instruction-level irregular parallelism is as described in Reference 1. This is derived from the examination of the nature of the inherent parallelism found in the small segment of software code that follows:

```
L := I1 * I2;
M := I3 * I4;
N := I5 * I6;
K := L * M * N;
```

This code multiplies the variables I1, I2, I3, I4, I5, and I6, and places the result in the variable K. The potential software parallelism can be found only by discarding the traditional view of a program. Instead of viewing a program as a list of instructions to manipulate data in fixed storage locations in a defined sequence, one must concentrate on the role the individual storage locations play as they temporarily hold data values that pass between operations in the program. This data dependency can then be described graphically as discussed below.

1. Data Dependence Graphs

The construction of data dependence graphs for a program provides a different view of the combination of data with operators. Optimizing compilers that are commonly used in conventional machines provide algorithms for this task. By drawing a series of arrows within the program segment above, one for each variable, with the head of the arrow showing where the variable is consumed and the tail showing where the variable is assigned, we obtain the graph shown in Figure 1.

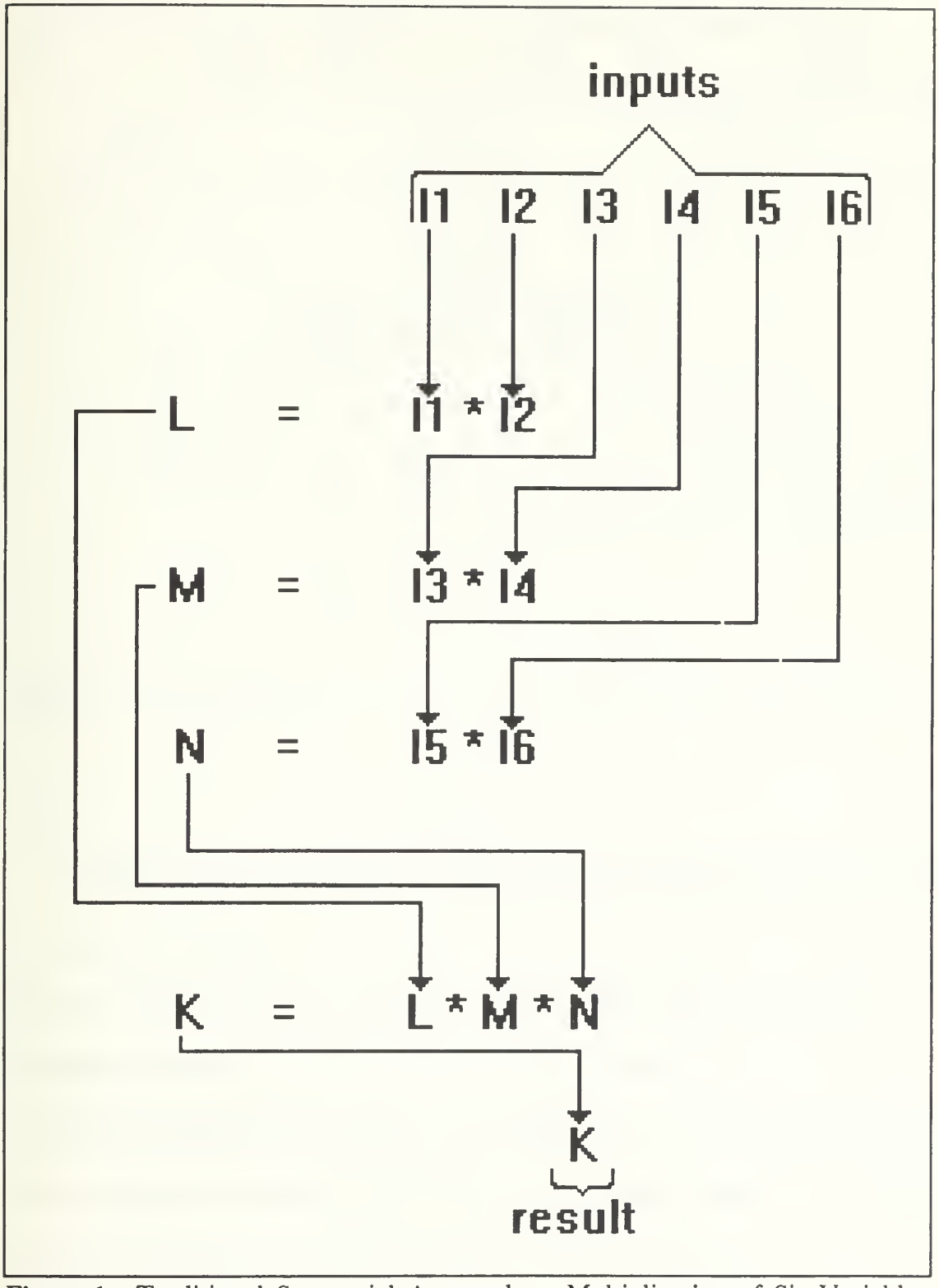

Figure 1. Traditional Sequential Approach to Multiplication of Six Variables. (From Ref. 1)

By rearranging the alignment of the diagram so that it shows potential concurrency across the page, we obtain the graph shown in Figure 2.

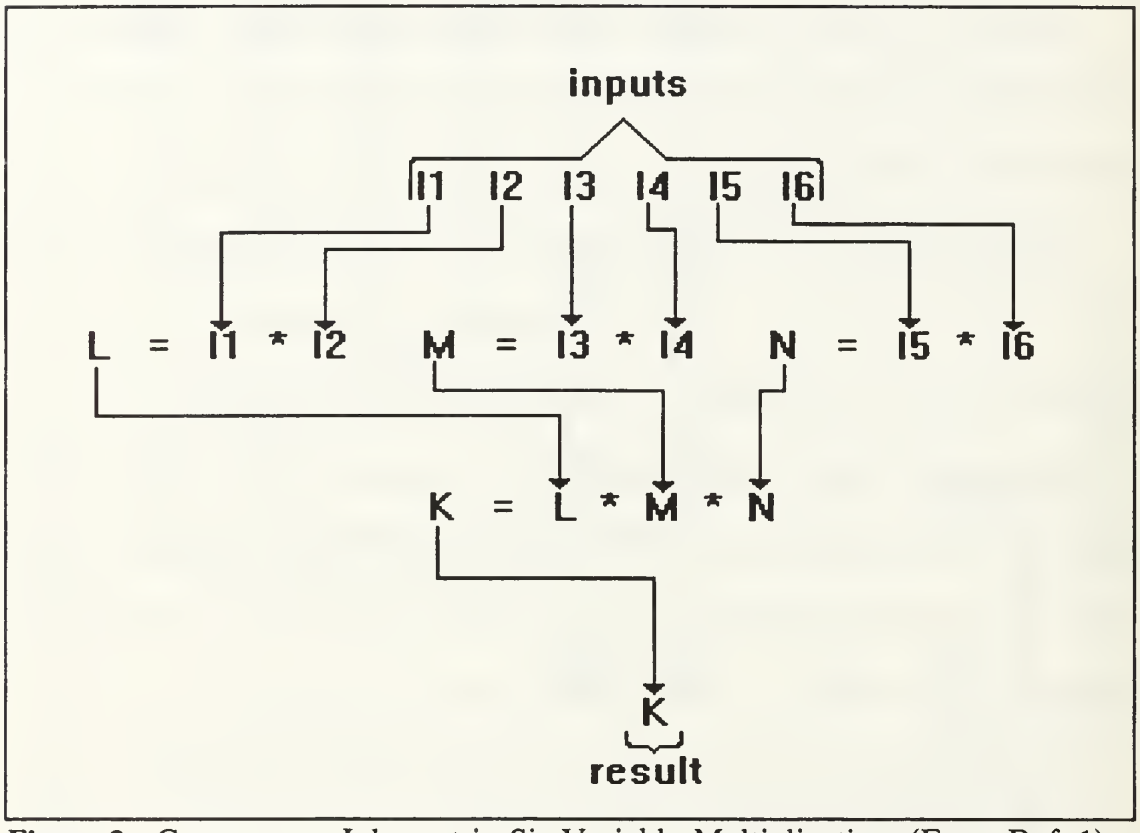

Figure 2. Concurrency Inherent in Six Variable Multiplication. (From Ref. 1)

As it is the goal to show that the inherent parallelism is only dependent on the data, elimination of the variable names within the structure of the graph yields Figure 3. The variable names are left in this graph for ease in readability by providing a simplified description of the expression to be computed.

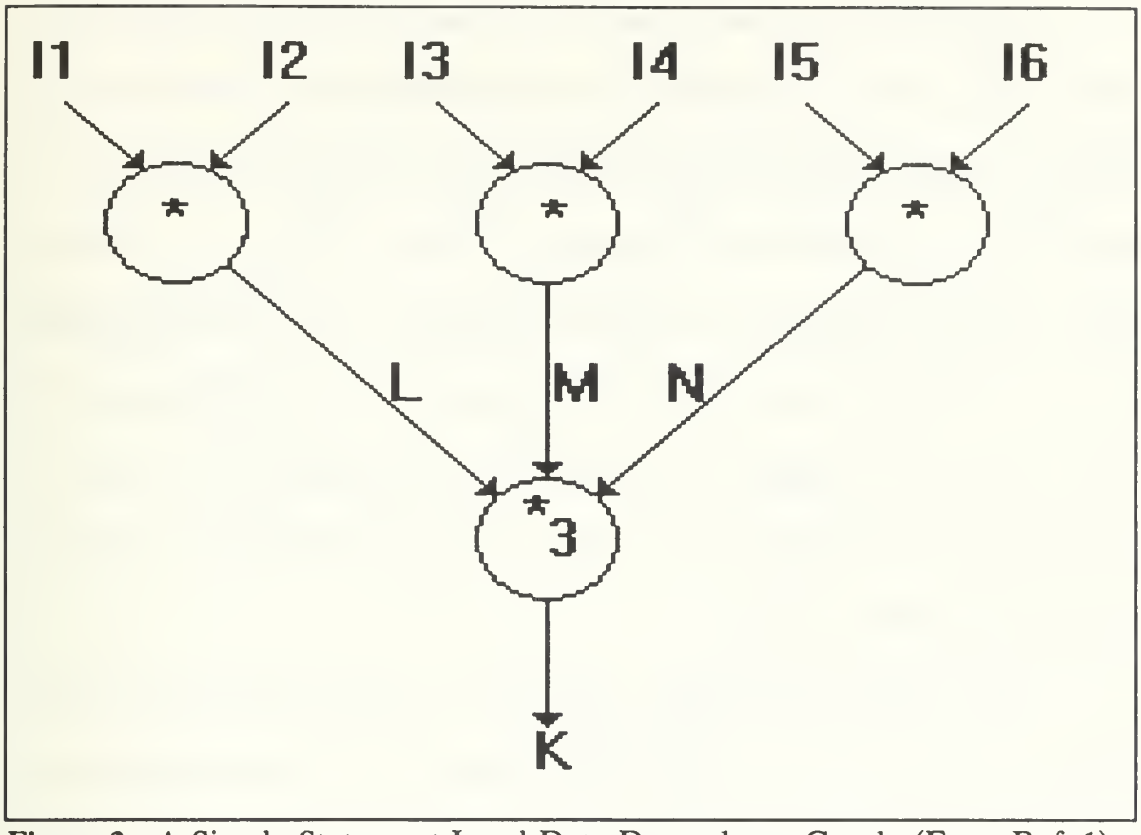

Figure 3. A Simple Statement Level Data Dependence Graph. (From Ref. 1)

The graph in Figure 3 is a final version of a simple statement-level data dependence graph. Not only does it retain the meaning of the original program block, it also shows the potential parallelism and enforced sequence in a two-dimensional format. However, Figure 3 does not fully illustrate all of the parallelism available for exploitation by instruction-level parallel hardware. In order to implement the algorithm shown in Figure 3, the availability of a system capable of multiplying three values together would be required.

Going on the assumption that the hardware implementation for a dataflow computer will use an instruction level commonly found in 16-bit minicomputers with extended arithmetic capabilities, an additional branch is required to eliminate the need for a three-value multiplier. This can be achieved by using a two-value multiplier to evaluate the variables L and M, and passing this result to a two-value multiplier along with the variable N in order to obtain the result K. It can also be achieved by passing the variable L along with the result of multiplying the variables M and N to obtain the result K. The same result will be obtained with either method.

The resultant graph shown in Figure 4 is a complete description of the available parallelism for the instruction block, using only two-value multipliers as discussed in the preceding paragraph. Also in Figure 4, each multiply instruction has been given an identification number. This notation is used to describe how all potentially concurrent instructions may execute simultaneously. From the sequential point of view, the order of multiplications would be (from Figure 4) {1}, {2}, {3}, {4}, and {5}; thereby producing the result in five multiplication times. However, the result may be obtained in as little as three multiplication times if sufficient processors are available. With two processors available, the order of multiplications would be {1,2}, {3,4}, and {5}. With three processors available, the order of multiplication would be {1,2,3}, {4}, and {5}. Due to the dependency of step four on the results from steps one and two, and the

dependency of step five on steps three and four, the final result will require a minimum of three multiplication times to be calculated.

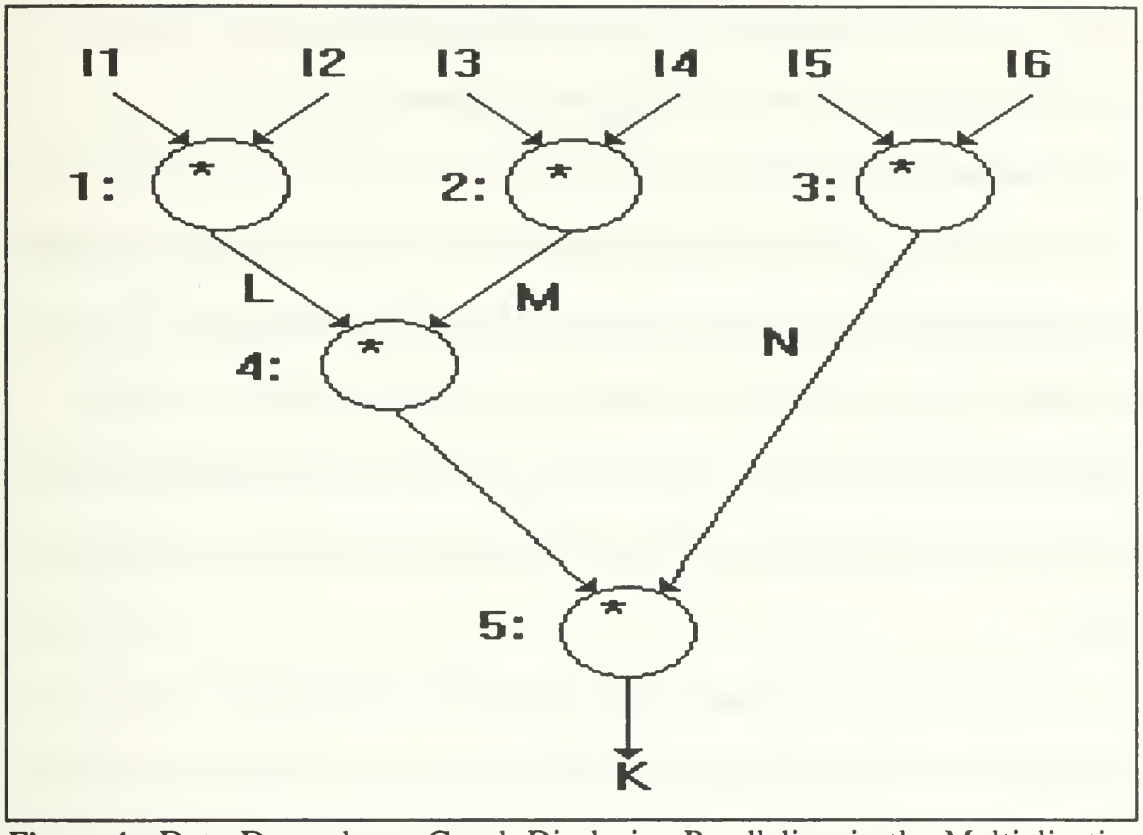

Figure 4. Data Dependence Graph Displaying Parallelism in the Multiplication of Six Variables. (From Ref. 1)

The key result determined from looking at Figure 4 is that this graph shows how the *instructions are dependent on the data.* As discussed in the preceding paragraph, the execution of step four is dependant upon the results determined in the execution of steps one and two. Thus, it would be safe to say that it makes no sense to execute an instruction before all data required is present. On the contrary, it shows that once an instruction has completed executing, all other instructions awaiting its output data can be safely executed.

Thus, Figure 4 shows that the simplest method of execution for a graph program is to send data directly from instruction to instruction, and to allow each instruction to execute only when it has all of the required input data. Hence, it can be said that graph program execution is data-driven.

2. Generalized Dataflow Graphs

The dataflow graph discussed in the previous section is not a good example of conventional computing practice. With no control structures such as conditionals or loops, and only one arithmetic operation, it does not completely represent what would be expected in general. This section briefly describes the enhancements to dataflow notation that will accommodate the more general program.

Since *any* form of machine instruction can be represented as a node on a data flow graph, it would therefore follow that any instruction can be executed in parallel with any other instruction. For this reason, graph notation is very useful in exploiting irregular software parallelism. The simplest case, demonstrated above, is in the evaluation of the general arithmetic expressions in which any arithmetic machine instruction is used. Additional parallelism is apparent with the introduction of control structures such as conditionals and loops, as well as the inclusion of functions and structured data.

a. Conditionals

The simplest control structure is the conditional (if..then..else). Data dependance graphs are constructed utilizing *conditional dependance arcs* that

are controlled by the runtime execution of a boolean expression. The implementation of these arcs occurs through the use of two 'switching' machine instructions; branch, and merge.

A branch instruction compares the data input value with the boolean control input value. The path selected depends on whether the boolean expression evaluates to true or not. Hence, it may be viewed as a switch, selecting one of two possible paths depending upon the controlling expression.

A merge instruction compares the boolean control input value with two data input values A and B. If the boolean expression evaluates to true, the output receives data value A, and if it evaluates to false, the output receives data value B.

As with the dyadic arithmetic instruction discussed above, instruction execution does not occur until all required input values are present. Also, since only one of two possible routes are selected, the other route is left inactive during execution of the conditional statements. It should be noted that the evaluation of the boolean expression and the selection of the proper path, can be performed concurrently with any other machine instruction.

b. Loops

Both the branch and merge instructions described above can be thought of as switches. These instructions are extremely powerful constructs when used to implement graphical loops. Graphical loops allow the definition of very

complex computations by relatively small programs, similar to that found in conventional loops.

The major difficulty experienced with looping constructs in parallel computations is the possibility of simultaneous activation of reentrant code. It is essential in practical dataflow systems to restrict the execution of an instruction until it is known that the output path for that instruction is empty. This requires that the paths of execution for a dataflow graph behave like first-in-first-out queues.

Systems permitting only the sequential cyclic reentrancy just described are known as *static* dataflow systems. Static dataflow systems are the simplest implementation for reentrant graph programs. However, they do not permit concurrent reentrancy as the loops must be reactivated in a strict sequence only. Limited parallelism is achieved through pipelining within the cycles of a loop, and the additional parallelism available is only achieved through the use of functions and/or structured data.

c. *Functions*

By considering a function as a user defined subgraph in a dataflow program, a subgraph that can be called from several places within the program, it is noted that concurrent reentrancy must be required. Although it is possible to create many copies of the machine code that represents the function, and to place those copies in-line where needed, this is wasteful of instruction storage when large functions are implemented, or when a function is called

several times within the program. Another fault is that this scheme implies the inclusion of infinitely expanded program graphs if recursion is to be utilized. To overcome these faults, the apply-exit scheme was developed. [Ref 1.]

This approach to function implementation is analogous to conventional macro-expansion in that extra code and data space is allocated only when called for, and then released when no longer needed. In this scheme, concurrent reentrancy is permitted via an apply instruction. This instruction is placed at the start of a user defined subgraph, and creates a new copy of the subgraph each time it is called. All input paths to a subgraph activation are gathered together and transferred to this unique new copy of the reentrant code. An exit instruction, occurring at the end of the subgraph, collects all of the output data from the subgraph, and transfers it back to the calling apply instruction. The copy of the reentrant code is then destroyed. Thus, data is not required to share code concurrently.

d. Structured Data

Through the use of a single variable name in referring to a large number of simple data items, compact computer programs can be written and executed. A separate storage area can be created to hold the structures, and the structure can be represented by a *pointer* when traversing the dataflow graph paths. A specialized *structure store* has the responsibility of issuing these pointers, as well as executing the read and write operations on these structures. All other

operations on the structures are as described in the preceding paragraphs for single data items.

3. Graph Code Compilation

It is apparent from the previous discussions that it is possible to generate dataflow graphs from conventional high-level programming languages. The difficulty lies in the analysis algorithm. This is the tool that forms the dataflow graphs from the conventional language, is highly complex, and requires a long time to execute. The solution has been for researchers to develop other languages that are easier to translate. Dataflow research projects are concentrating on these new languages.

a. Conventional Languages

The conventional language programmer accesses variables during the program execution. It is this explicit use of storage locations that produces possible side-effects in dataflow development. Due to the obscure expressions used to index arrays, compile-time data dependence analysis is difficult, and requires programmer intervention in specifying how arrays are to be accessed. Another difficulty lies in the use of unbounded arrays and pointer arithmetic. These language features are impossible to decipher and compile-time analysis can never occur. It is best that these facilities of conventional languages never be utilized in the dataflow development process.

b. Single-Assignment Languages

These languages allow each variable to be assigned only once within a program. This scheme helps to eliminate the ambiguities that may arise when reassigning values to variables. Single-assignment languages (SALs) utilize no direct control statements, nor do they allow for sequential execution. However, SALs do have provisions for permitting controlled reassignment of variables used in loops, and other special cases.

The use of arrays, streams, and other data structures that can be readily implemented in dataflow graphs are widely utilized by SALs. The use of these structures does require strict typing and scoping rules, such as the prohibition of all types of side-effects in the reentrant construct. These features result in languages ideal for the syntactical description of dataflow graphs. Most of the single-assignment languages developed are done so without reference to dataflow execution, but may refer to other methods of execution.

c. Functional Languages

These languages are those that have been developed without reference to any particular means of execution. They are based on the mathematics of functional algebra. Functional languages have no provisions for storage states. They differ from SALs in that they have no concept of assignment. They are referred to as *zero-assignment* languages in which variables are defined instead of assigned.

Similar to SALs in the absence of control statements and side-effects, functional languages are more powerful. The basis on functional algebra permits the construction of abstract data structures and higher order functions, thus making them more powerful than SALs. Although these two groups are not directly equivalent, both have enough in common to make it attractive for implementing functional languages on dataflow systems. SISAL has features of both languages, as shall be discussed in following chapters.

4. Dataflow Graph Summary

The translation of dataflow graphs from a wide range of high-level programming languages is very feasible. They permit data structures, functions (with recursion included), loops, and conditional control constructs, thereby providing a convenient notation to represent parallel computations.

III. PROGRAMMING AND FUNCTIONAL LANGUAGES

A. INTRODUCTION

Based on the algebra of functions, the origin of a functional language can be traced to the development of *lambda calculus* by Alonzo Church in the 1930's. This calculus arose from the attempt to identify those functions on positive integers that could be computed in a purely mechanical or algorithmic method. Church proposed that these effectively calculable functions be expressed in a simple calculus, the lambda calculus. Universally regarded as true, the lambda calculus provides a good basis for the design of a programming language.

In the lambda calculus, functions are denoted by expressions known as *lambda expressions*. As derived from Referance 1, the expression:

$$\lambda x.\ x^3 + 8$$

denotes the function that cubes a number and adds 8 to it when applied to that number. It can be seen that the lambda expression is composed of two parts. The bound variable is that part to the left of the dot, and the body is that part located to the right of the dot. Abstraction is the process of bringing the two halves together. Hence, the function denoted by the lambda expression is abstracted from its body.

The application of the function denoted by a lambda expression occurs by juxtaposing them with their argument. Therefore:

$$(\lambda x.\ x^3 + 8)(2)$$

indicates the application of the denoted function to the value 2, and evaluates to 16.

Another method is for the lambda expression to appear in the argument position. Note that in the following case the bound variable f is 'function valued':

$$(\lambda f.\ f(2))\ \ \lambda x.\ x^3 + 8$$

and once again evaluates to 16.

Both lambda expressions described above form the well formed formulas of the lambda calculus. This calculus is then completed by the set of rules for lambda conversion. These rules convert one lambda expression to another without a change in the meaning of the expression. As these rules are purely syntactical in nature, it is not necessary to understand the meaning of an expression when applying them.

There are three rules for lambda conversion [Ref. 1]:

- Alpha rule: The name of the bound variable may be changed as long as it is done so consistently throughout.
- Beta rule: A lambda expression of the form $(\lambda x.M)\ N$ may be converted to the form M[N/x]. That is, M with N substituted for x, as long as it is done so consistently.
- Gamma rule: An expression may be converted to an abstracted function reapplied to appropriate arguments.

The computation of a function corresponds to the application of these rules, as they can be applied mechanically. Essentially, this is the idea of the lambda calculus. Reduction is the term associated with the application of the beta rule.

The conversion of some equation A to some other equation B by only applying the alpha and beta rules implies that A is reducible to B. An expression is in normal form if it is no longer reducible. Normal form lambda expressions are unique, and correspond to the result of program evaluation.

The reduction of a lambda expression may follow one of several different paths. The *Church-Rosser theorem* is the main theorem of the lambda calculus. This theorem states that the order of reduction is unimportant, as all paths lead to the same result. It is the result of these rules and theorems that make the lambda calculus an amenable solution to the need for computational formalism.

B. FUNCTIONAL PROGRAMMING

It is not surprising that the object of a functional program is the definition of a set of functions. However, there are fundamental differences in the functions defined in a functional programming language, and those defined in conventional languages in which functions can be defined (i.e., Pascal).

As discussed in Referance 1, a program written in a functional language consists of a set of equations. These equations define functions in terms of other functions that are simpler, or primitives to the language.

For example, consider the following definition of a maximum function:

max(x, y) := **if** x > y **then** x *else* y

This function is defined in terms of the well known primitives > and if-then-else. A program which uses this defined function in the definition of another function may have the following form:

```
maxthree(u, v, w) := max(max(x, y), z)

max(x, y) := if x > y then x else y
```

It should be noted that there is no ordering implied. This implies that the execution of a functional language program is accomplished by evaluating the equations of the program as directed. This is demonstrated in the following step by step analysis of the previous program segment to find the maximum of the three numbers 2, 7, and 9.

```
maxthree(2, 7, 9)
=> max(max(2, 7), 9)
=> max(if 2 > 7 then 2 else 7, 9)
=> max(7, 9)
=> if 7 > 9 then 7 else 9
=> 9
```

As in conventional languages, functional languages provide for recursion by allowing functions to be defined in terms of themselves. Consider the following well known definition of the factorial:

```
factorial(x) := if x == 0 then 1 else x * factorial(x-1)
```

This definition may also be expressed by the following set of equations:

```
factorial(0) := 1
factorial(x + 1) := (x + 1) * factorial(x)
```

Once again, ordering is not of importance. This requirement is obtained by ensuring that at most one equation applies for any value of input.

1. Qualified Expressions

An important property of functional languages is that within a given context, the same expression evaluates to the same result. This is required due to occurrence of repeated subexpressions on the right hand side of equations. The following expression demonstrates this phenomena:

```
g(x) := if x = = 0 then 0
     else x + (g(x/2) * g(x/2))
```

There is no change in the meaning of the program using this method of expression. However, problems could arise in the efficiency of program execution if (as in this case) the repeated subexpression occurs in a recursive call.

Functional languages provide for this type of inefficiency. The use of *qualified expressions* allow a programmer to name the repeated expression, and then refer to it by that name when using it. An example found in many functional languages is the *where* construct, such as:

```
B where y = A
```

In this sense, the variable named y is used to refer to the expression A throughout the evaluation of B. Therefore, our example above would be written:

```
g(x) := if x = = 0 then 0
     else x + (y * y) where y = g(x/2)
```

It must be noted that the meaning of any expression involving the where construct is always equivalent to the original expression when the qualified variable is resubstituted back to the expression which it denotes. Thus, in the

above case, the value of y *does not* change throughout its use. Therefore, the where construct is **not** an assignment expression.

2. Higher Order Functions

The term *higher order* implies that within a program, the functions themselves are passed around as data objects, similar to the passing of scaler and lists. All realizable functional languages must have this property, thus ensuring that all objects are treated as equals.

This implies that functional languages contain expressions that, when evaluated, return function valued objects. An example of this is the factorial program. In the set of equations defining the function factorial above, the identifier factorial (which in this case is the data object) takes as its value the factorial function.

It is this existence of the higher ordered function in a functional language that provides for a powerful programming style. It is, therefore, possible to define general purpose iterators that traverse a data structure, and apply functions which are passed as parameters within the program. Provided with enough high order functions, explicit recursions may be omitted from programs by instantiating these functions where needed.

3. Data Structures

Within a functional language, provisions have been made to allow handling of data structures through the use of additional functions called *constructor functions*. A constructor function is not defined by a set of equations,

it has the sole purpose of building data structures. Actually, the terms created by constructor functions and constants (which may be viewed as unary constructor functions) name the data structures.

The use of constructor functions allows the creation of simple data structures. Once a structure has been defined, the programmer would then write equations defining a function over the structure, just as if writing equations to define a function on a scaler.

A modern functional language would have provisions for the three concepts just described, as well as have an ability to allow for set expressions, data typing, modular structures, etc. Those features described above would be considered the main features necessary for a functional language. The basic concepts developed from these features, and the consistent employment of those concepts, form powerful notations that characterize functional languages. Once a programmer has grasped the basic concepts of the language, learning the functional language becomes easy. Another feature is that a functional language is *deterministic* in that the same result is always obtained for a given input. As a functional language may provide for alternative sequences of evaluation, it must produce the same result if terminated correctly.

C. FUNCTIONAL PROGRAMMING IMPORTANCE

It is widely held that a functional language is more powerful than a conventional language, thus allowing for simpler program construction with fewer errors occurring in the task. Additionally, since a functional program allows for

formal manipulation, it enables the process for program transformation, which is the systematic derivation of efficient programs from the program specifications. Third, it is easy to organize the parallel evaluation of a functional program. This provides for the design of very fast, efficient, expandable, multi-processor machines. It is these three claims that provide arguments for why a functional language may be considered more important, with significant advantages over conventional languages.

Since the execution of a program, written with a functional language, depends only upon the meaning of the component subexpressions and not the history of any computation up to the evaluation of that expression, the program may be viewed as a static object. Thus, functional programs are referentially transparent. As there is a clear notion of the equivalence between expressions, one expression may be substituted for another without changing the meaning of the whole expression in which the equivalent expressions are used. As an example, mathematics is referentially transparent in that one may substitute the expression (7 - 3) for the expression (4) when used in the expression (5 * 4) without changing the whole meaning. In this case, (5 * (7 - 3)) = 20 = (5 * 4). This demonstrates that an inherent ground rule for the notation used to write programs must be comprehensible, and manipulable. Both of these features are found in functional languages.

A conventional language program is written with the extensive use of variables. The meaning of any expression that depends upon these variables will

vary according to the history of computations using that variable prior to the evaluation of that expression. As noted above, the use of variables in a functional program is treated only as if defining that variable, the variable would not change during the execution of the program.

This frees the programmer to concentrate on *what* the program is to do, not *how* the program is to do it. In functional languages, the output of a program is independent of the order of evaluation, whereas conventional programmers must concentrate on the order of execution to ensure the proper result is attained.

1. Specification

The use of a functional language allows for the programmer to prototype, or specify the algorithm prior to developing the actual program. This process, in turn, lends toward the development of more efficient programs. Functional languages enhance the programmers ability to design a model of the desired program. Since the specification and the program are written with the same notation, this allows ease in testing, or demonstration of an expected capability.

Recall that functional languages are often viewed as a subset of a general equational language, only a language that allows the equations to be used as directed rewrite rules. During the development of the language, restrictions are often placed on the notation of the language to promote efficient interpretation of programs. Removal of some or all of these restrictions would

allow a user to define functions using more general equations, and thereby improve the power of a functional language for specification purposes.

2. Transformation

Once the necessary specification is established, an efficient program to accomplish the desired task must be developed. Transformation is the concept in which the specification is systematically manipulated to produce this program. Thus, it is critical to define a set rules for manipulation which leave the meanings of programs unaltered while transforming them to improve their efficiency. One of the great advantages of functional languages is that such a set of rules can easily be provided. Due to the inherent fact that functional languages are referentially transparent, it allows them to be manipulated in a similar fashion to the manipulation of mathematical forms. The ability to completely interchange equivalent expressions, without the need for elaborate checking, lends itself to the development of simple rules for transforming functional programs. Conventional languages do not lend themselves to these transformation rules as freely because they require strict adherence to methodology and program flow.

It is this formal transformation of a functional language that opens the door to at least a partial mechanization of the process. Thus, a user may write a structured transformation plan with high level transformation operators which will aid him in the design of his program. What this does is provide for the semi-automatic development of programs by utilizing the computer to aid in software

design. Ultimately, this will lead to higher standards of accuracy, reliability, and reproducability within the software development industry.

3. Parallel Execution of Functional Programs

Since the utilization of a function in programming allows for the development of side-effect free code, functional languages lend themselves remarkably well for parallel execution. Within the definition of a function, the determination of multiple values would be indicated by a comma separated list of expressions. These individual expressions lend themselves freely to parallel execution by a dataflow system. Additionally, the absence of side effects allows the subexpressions to also be computed independently.

It was observed in the development of the dataflow model discussed in the previous chapter that the graph structure portrays itself as an attractive model for a parallel processing system. By considering each node as a process, and each path between nodes as a communication channel between processes, it is possible to view a common distributed system in graphical form. An extension of this would then be to consider a function defined within a program written in a functional language as a process. It is apparent, then, that a functional language lends itself freely to parallel execution of programs.

D. FUNCTIONAL LANGUAGE APPLICATIONS

The potential exists for universal applicability with a functional language. It has the ease of programming style associated with conventional languages while providing for modularity within the programs written. It is thought that these

general purpose programming languages will one day replace the sequential languages widely used today for scientific programming.

While opening the doors for providing applicative parallel languages of the future, functional languages are able to be utilized on today's sequential machines. Much research within the utilization of functional languages for both sequential and parallel processing is being carried out today.

IV. LOOSELY COUPLED DISTRIBUTED COMPUTER SYSTEMS

A. ARCHITECTURE

The connection of two or more computer systems via a communication link, as is shown in Figure 5, with each system having its own operating system and own storage facility, yields what is termed a *loosely coupled multiprocessing* system. [Ref. 3] Each system within the interconnection is allowed to operate independently, and can communicate with the other systems when necessary. Communication between systems occurs via message passing and/or remote procedure calls. This communication occurs at the input/output level. The separate systems are allowed to access each other's files, and can, in some instances, switch tasks between lightly-loaded processors to achieve some modicum of load balancing. Logically, one may view a loosely coupled system as a collection of processes running on various processor elements. Processes running on the same element are allowed to communicate using shared memory, while processes running on different elements must communicate via messages.

Local area networks (LANs) are, essentially, the backbone for loosely coupled systems. A LAN is the interconnection of two or more computers via coaxial cable, fiber optics, etc. while providing a means for intercommunication between systems.

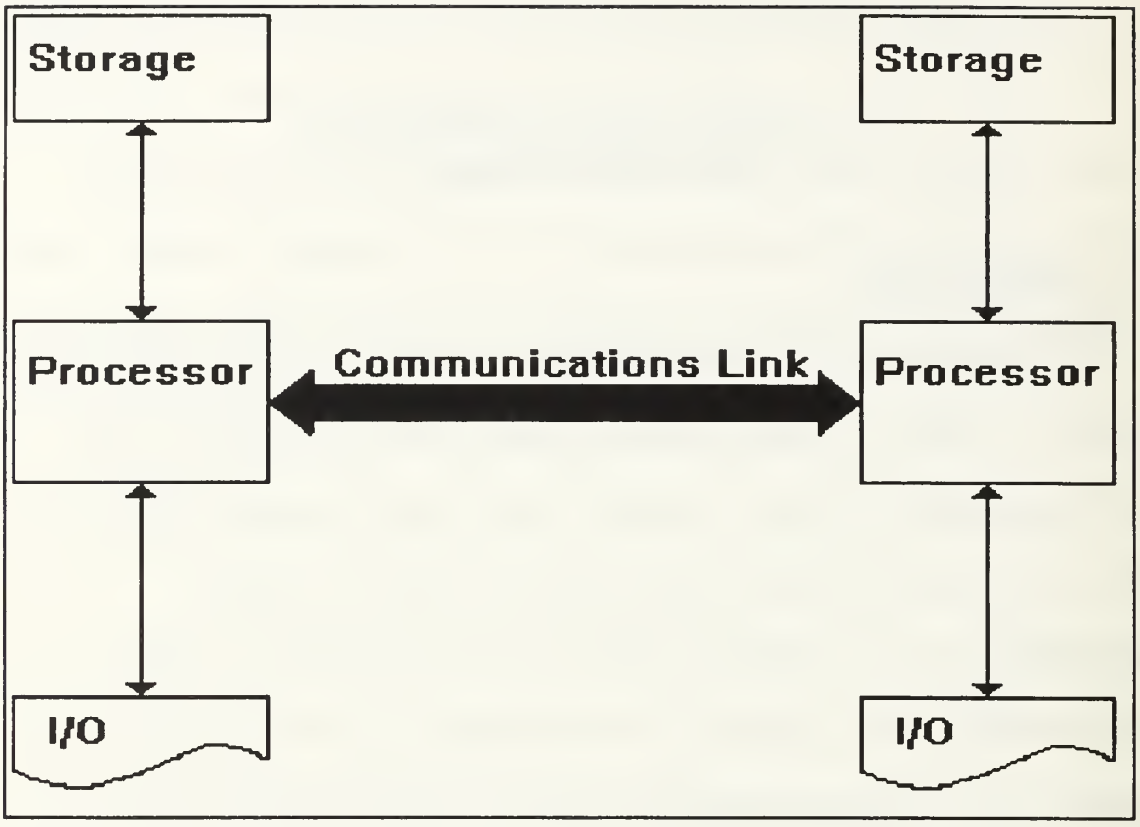

Figure 5. A Loosely-Coupled Multiprocessing System. (After Ref. 6)

1. Objectives

What is expected by the user of a distributed computer system? The main objectives in the development of loosely coupled systems, as derived from Reference 3, are to provide for the following.

- *Increased performance.* Attained through the judicious use of the multiple processing elements present. The user must ensure that they allow for contentions and bottlenecks in system operation when using a shared bus, but the overall system performance can be improved.

- *Extensibility.* Provides for a simpler system design, installation ease, and overall ease in maintaining the system. It is inherent that these systems be able to adapt to changes in the environment without overall design changes. Modifications to the performance requirements and changes in the functional requirements shall not affect the overall system.

- *Availability.* With the increased reliability of hardware it has become possible to obtain duplicate and triplicate systems for backups. Further research in software is also providing for improvements in its reliability and making it more readily available. The judicious combination of these aspects of the computer industry ensures that users will achieve a more reliable system.

- *Resource sharing.* The resource may be something as trivial as a peripheral device, or something as complex as a file sharing system. These systems also provide for the static and/or dynamic allocation of the resource to be shared.

B. SUN NETWORK OF WORKSTATIONS

The ECE Department's Computer Network is a collection of nearly 40 stand-alone SUN workstations. These workstations are then connected together through the use of coaxial cables, and set up into a network utilizing the TCP/IP protocols for inter-workstation communications. Additionally, the ECE Department's Network is built around four primary file servers, with an additional file server dedicated to the Digital Signal Processing research areas. Each of these file servers has a printer associated with it for use within the network. It is this interconnection of workstations, file servers, peripherals, etc., that may be considered as a loosely coupled multiprocessing system.

The Transmission Control Protocol/Internet Protocol (TCP/IP) is a common-name used to describe a multitude of data-communications protocols. These protocols are used to organize computers and other data-communications equipment (i.e., faxes, etc.) into computer networks. The TCP/IP incorporates the file transfer protocol (ftp) and the TELNET protocol for inter-workstation communications.

When working on the net, the user will be at a dedicated "local workstation". The user then has the ability to access files from the file server(s), from remote workstations, and share other peripheral devices connected to the net. The user also has the ability to use the network for over-the-network execution of commands on a remote workstation. This ability can be achieved through the use of rlogin, where the user remains at the local workstation, but is logged into the remote workstation. Additionally, the SUN Operating System provides a facility for use of the remote processor. Through the use of a remote shell command (rsh), the operator would have the capability of using the remote workstations processor for execution. It is this capability of the SUN Network that this research is attempting to exploit.

V. IMPLEMENTATION OF THE SISAL LANGUAGE ON THE ECE NETWORK OF SUN WORKSTATIONS

A. OVERVIEW OF THE SISAL LANGUAGE

SISAL (Streams and Iterations in a Single Assignment Language) was developed as a functional language for parallel numerical computation. The project began in 1982 with the following goals defined in Reference 4:

- To define a general-purpose functional language capable of efficient operation on both conventional and parallel architectures.
- To define a dataflow graph intermediate form. This intermediate form shall be independent of language and target architecture.
- Achieve sequential and parallel execution performance competitive with programs written in conventional languages.

The developers of the SISAL Language have been able to demonstrate the dataflow approach to computing as a viable option outside the classic dataflow world, and without the use of specialized hardware. The SISAL Language has successfully operated on several different system architectures, including the Cray-X/MP, the Manchester dataflow machine, other shared-memory multiprocessor systems, and uniprocessor systems. As the research continues in the area of development of the system, other experimental systems come into play.

Much of the development of the SISAL language has been directed towards applications on uniprocessor systems (i.e., stand-alone workstations) and on closely coupled multiprocessor systems where several processors are connected

together, and share a common memory. It is the goal of this thesis to attempt to exploit the parallel processing capabilities of the language, and to implement them on a loosely coupled multiprocessor system.

1. SISAL Program Structure

The executable entity created by SISAL consists of one or more separately compilable *compilation units*. These may include a program, an interface, and object modules. The program is the simplest entity. It may contain some type declarations, some user defined functions, declarations that may define other entities to be imported from other modules, and other definitions. It is usually one of the functions within the program that is the starting point for execution. As this is the outermost level of the executable item, function parameters and results are handled through communication at the operating system level.

Although a module is similar to a program, it contains no provisions for starting execution. Modules pair with interfaces for exporting type and function names. Thus, the module and the interface are related, with the interface being given the same name as the module.

There are two types of interfaces. Those associated with modules as described in the preceding paragraph provide a means for the declaration of public functions (universal scope of use). It is through this vehicle that these functions are allowed to be accessed by software not written in SISAL. There are also stand-alone interfaces, which are used to declare the relationship between

SISAL and a set of subsidiary code written in other languages. Thus, SISAL software has access to other language libraries.

2. Data Types

The SISAL language has the ability to handle a rich set of data types. They include the usual structured types such as records, arrays, streams and unions, as well as the usual scaler types (integer, character, boolean, real, etc.). The constituent components of a structured type may be of any other type.

Within the SISAL language there are occurrences where function values are parameters to other functions. They also can be the result of expression evaluation. Hence, a function type is allowed as a declared type as long as the types of all parameters and results are given.

3. Functions

As a functional language, one of the most important topics to be discussed is the function. The declaration of a function is composed by listing the name of the function along with the names and types of formal parameters associated with the function, and the type of the result value(s). A function then contains one or more expressions with types corresponding to those listed in the result types. The values utilized by the function are accessed through the formal parameters, not through globally accessed names.

4. Expressions

The foundation of any functional programming language is the expression. It is a single expression, or the combination of a group of expressions,

that comprise a function definition. Within the SISAL language, the syntax for any expression was designed for familiarity and clarity.

a. Simple Expressions and Name Scoping

The SISAL language provides predefined functions for type conversion. In addition to this, SISAL supports type promotion (i.e., promoting a real value to a double-real value). Also, within SISAL, the conventional operators are used for combining scaler arithmetic values.

Name scoping is via qualified operators as discussed in Chapter III. Hence, the value of any expression may be assigned to a name using the **let** construct. Once this renaming has occurred, the name may be used in place of the expression within the scope of the definition.

b. Arrays

The SISAL language provides a rich set of operators for array definition and manipulation. As with other languages, there is defined a useful set of functions on arrays, and conventional arithmetic operators may be used for element by element operations. Through the use of vector subscripts, arbitrary sections of the array may be addressed.

c. Streams

A sequence of values produced in order by the evaluation of one expression, and consumed in the same order by one or more other expressions is a stream. The usual producer and consumer expression is of the for construct, but other forms are also available. In SISAL, the ability for the consumer

expression to start before the producer expression is finished exists, thus allowing for the pipelined parallelism that streams make possible.

5. Selection Control

The SISAL language provides two vehicles for selection control: the **if** expression and the **case** construct. The paths followed during an if statement are determined by the evaluation of a boolean expression. The paths followed during a case construct are determined by the value(s) of the selecting expression (similar to the switch construct found in the C programming language).

6. Iteration Control

The use of the simple **for** construct within SISAL provides two forms for potentially parallel and sequential evaluation. The first form distributes values to the bodies of the construct. Each body then defines values that contribute to the overall result. The second form generates dependencies between values defined in one body and used in another. Either form allows for the values to be collected within an array, in a stream, or reduced to a single value.

7. Error Handling

The management of erroneous computations occurs in SISAL through the use of standard error processing semantics. Each SISAL type has a defined **error** value associated with it. The language allows an error value to propagate through to program completion, as well as electing to stop program execution when an error is detected. By allowing the error to propagate through to

completion, the language provides a facility for allowing the programmer to search for and correct possible run time errors.

B. THE OPTIMIZING SISAL COMPILER (OSC)

Since the inception of the project in 1982, there have been several revisions of the SISAL compiler. The research performed in this study concentrated on Version 12.0 of the Optimizing SISAL Compiler. This compiler has been used on the Manchester dataflow machine, the Cray X-MP, Cray Y-MP, and Cray 2. Additionally, this SISAL compiler has been demonstrated to operate on sequential machines, such as a SUN Workstation operating the SUN Operating System (a derivative of UNIX).

1. Compiler Overview

The following is a brief overview of the OSC. Further discussion on this compiler is available in Reference 5. This section introduces the phases and subphases of the compiler. As shown in Figure 6, the separate modules used to make up the compiler are the SISAL parser, the IF1LD, IF1OPT, IF2MEM, IF2UP, IF2PART, and CGEN modules, as well as the host machines C compiler. The OSC will process the source code sequentially through each of these modules in order to generate the required executable entity.

A SISAL program is located in a file with the .SIS suffix, as written by the programmer. The first phase of program compilation is to convert this code into an intermediate form known as IF1. This intermediate form defines dataflow graphs that adhere to applicative programming semantics. These graphs are

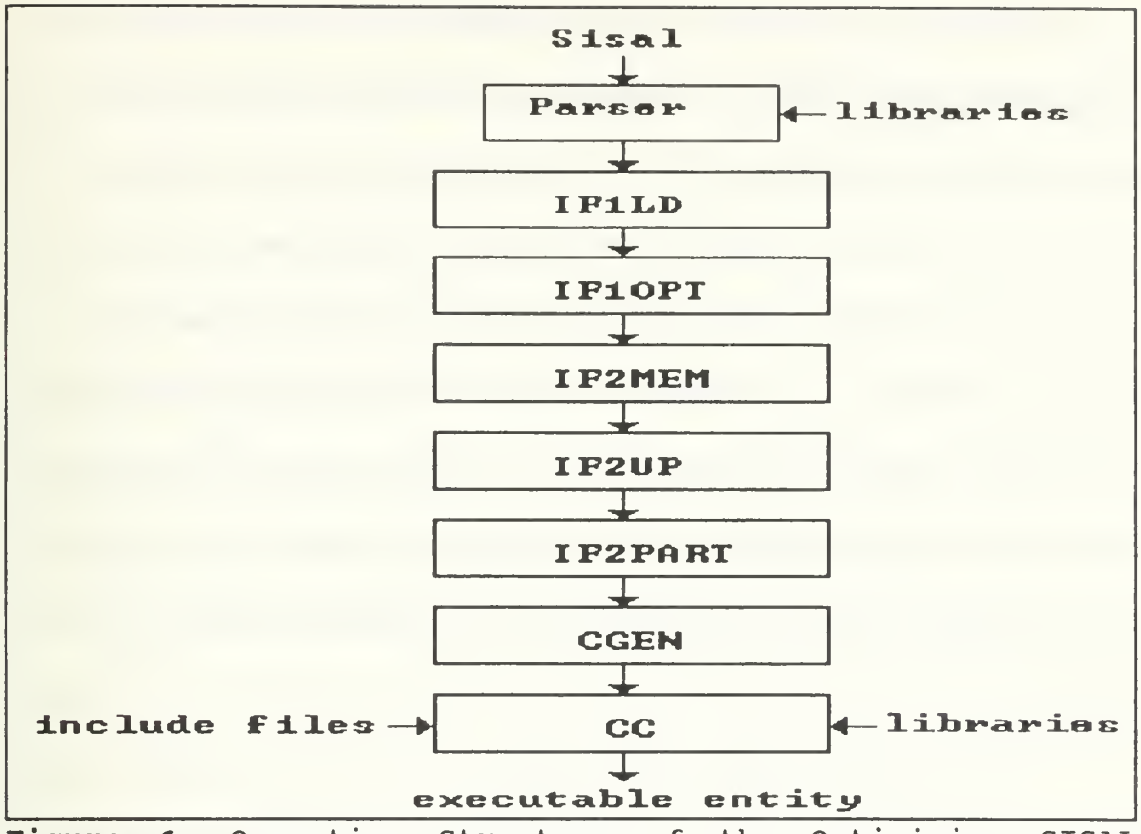

Figure 6. Operating Structure of the Optimizing SISAL Compiler. (After Ref. 5)

composed of simple nodes, compound nodes, graph nodes, edges, and types. The graph nodes denote operations, the edges transmit data between the nodes, and the types describe the data to be transmitted. Simple nodes are used to represent arithmetic operations and array and stream manipulation while compound nodes control one or more subgraphs that define structured expressions (i.e., conditionals, for loops. etc.).

Prior to this first phase, the source code is run through the C preprocessor for macro expansion and file inclusion. This allows the programmer to configure the programs compilation, as well as define and expand macros for inclusion in the program.

After the intermediate form of the program has been generated, it is sent through the IF1LD module. This module links the generated IF1 files to form a monolithic, or complete, IF1 program for further processing. This monolith is then read by IF1OPT which is a machine independent optimizer. Conventional optimizations, such as function expansion, common subexpression elimination, dead code removal, etc., occur in this module. By default, at the end of optimization, all functions except for those presented with recursive calls, are in-lined to form a single dataflow graph. Additionally, common subexpressions are also eliminated in expressions located outside the conditional statement branches.

This optimized dataflow graph is then sent to the build-in-place analyzer IF2MEM. This analyzer will preallocate array storage allowing compile time analysis. This permits the run time expressions to calculate their sizes. The overall result is a semantically equivalent dataflow graph in IF2, a superset of IF1 that is not applicative in nature in that it allows operations which directly reference and manipulate abstract memory. This IF2 program is then sent for update in place analysis during the next phase of compilation, which occurs in the module IF2UP. This module restructures the dataflow graph to help identify, at compile time, those operations that may execute in place during runtime.

It is this optimized, monolithic program that is sent to the parallelizer called IF2PART. This module will define the desired granularity of parallelism, with analysis based on execution time estimates. This module only selects for

expressions, and stream producers/consumers for parallel execution. Although the cost (overhead for small functions) remains to high for actual parallelization of the programs functions, the included **for** expressions and stream producers and consumers are selected for parallel execution.

Once the program has been parallelized, the dataflow graph is then passed to the CGEN module for the generation of C code. This code is then compiled by the local C compiler. It is then linked with applicable library software to provide support for parallel execution, storage management, and user interaction. The executable output file is then produced. Once again, the C preprocessor is invoked to allow the definition of target dependent operations and values.

2. SISAL Runtime System

The OSC provides runtime software libraries that support the parallel execution of SISAL programs, implements data structure operations, provides dynamic storage allocation, and interfaces with the machine's operating system for command line processing and input/output.

The support provided by the SISAL runtime system makes modest demands of the host operating system. Two queues of executable tasks are maintained and controlled by the runtime system: the ready list and the for pool. The ready list is composed of a list of those processes ready for execution with the next available processor.

Execution of the SISAL executable entity created by the compiler begins with the function main. At initiation, the formal parameters and runtime options for the function main are read from the command line. Non-stream input values are read during program initiation, and output results are written at termination. Stream parameters and results are associated with files, and special stream producing (input) and consuming (output) threads are added to the ready list for processing during execution. Although the runtime system allocates stacks for threads on demand, every effort is made to utilize stacks previously allocated and no longer required. This helps to minimize stack allocation and deallocation overhead.

a. Threads

During the initial phases of execution of a program, a command line option specifies the number of operating system processes to instantiate for the duration of the program. Called workers, these processes are constant in number during execution. These workers look for work to do in the form of threads, and the number of threads varies over the execution of the program.

A worker is held in an idle state until it is engaged in work provided by SISAL execution. The threads assigned to a worker may be from the ready list, a piece of a for expression, or return to the idle pool where it may be utilized to process any storage deallocations. If there is no for expression pool work, no ready list pool work, and no storage deallocations to process, the worker remains in the idle pool until the next available thread arrives.

3. Storage Management

One of the runtime system libraries included with the OSC handles dynamic storage allocation/deallocation. This is required by the compiler to provide support for arrays, streams, and for internal objects and stacks. This dynamic storage allocation permits a free worker to select a pointer associated with a block of storage from a list of free blocks, and thus minimize contention between workers.

C. SISAL INSTALLATION ON THE SUN WORKSTATION

The OSC software was obtained by anonymous ftp from Lawrence Livermore National Laboratory. These files were placed in the /cad.exp/sisal directory on SRVR 2 of the ECE Department's SUN network.

Included in this set of files was an executable shell script used for OSC installation. This script asks a series of questions concerning the configuration of the target system, and then it generates a Makefile in accordance with the configuration specified. A copy of this script, and the Makefile created for SUN installation, is included in Appendix A.

1. Single Processor Installation

Although it was the goal of this thesis to determine the feasibility of parallel processing with SISAL on the SUN network, it was necessary to ensure that there was a fully operational compiler prior to any experimentation. The Makefile created by sinstall was run, and the compiler installed in accordance with Reference 5.

Included in the files obtained above, and associated with the OSC, was a set of example programs. These files are located in the /cad.exp/sisal/Examples directory. One of these files, sum.sis (Appendix B) located in the /Sum subdirectory, was compiled, and executed as described in the included README file. The output was as expected, and the data file created indicated a processor efficiency of 98.6% during execution of the program. Hence, it was determined that the compiler worked satisfactorily.

In order to obtain a better understanding of the SISAL Language, the file fact.sis (Appendix B) was written and compiled. This is a small program composed of a single function to determine the factorial of the input integer. Once again, upon execution the output achieved was as expected.

2. Production of Multiprocessor Code with SISAL

Once it was determined that the OSC would work satisfactorily on the SUN workstation, investigation into the multiprocessing capabilities commenced. It was determined that a likely place to start would be to attempt to install the compiler using the Makefile created by sinstall, only changing some of the responses to attempt to generate code for a known parallel computer-multiprocessor. It was determined that a Cray X-MP would be a likely candidate, simulating two processors during the installation. It was hoped that the differences in the installed executable code would give an indication of where in the compiler source code to look to find where the code needed to be modified in order to allow for the utilization of multiple processors on a SUN Network.

The Makefile created by the sinstall shell script for the Cray X-MP, with two processors is shown in Appendix A. As expected, this Makefile differs significantly from that generated for the SUN. This is mainly due to the fact that the Cray is based on the UNICOS operating system, and the SUN utilizes the SUN OS (similar to UNIX) operating system. The primary difference lies in the different C compilers on each host. Additionally the UNICOS operating system has a different name for their library archiver and library randomizer.

An attempt to create a compiler capable of running on a SUN and producing C code for a multiprocessor Cray appeared to be successful. The Makefile was modified to incorporate the SUN C compiler, and the SUN operating systems library archiver and randomizer. Attempts to run this Makefile failed initially due to C preprocessor directives in the source code for some of the Backend Modules (IF1LD, IF2UP, etc.). These files contain a C program (if1(2)timer.c) for controlling the system clock in order to generate statistical data during program execution.

One of these preprocessor directives created a timer named identifier only if the host machine defined was **not** a Cray. Once again, the goal was not to install a SISAL compiler for a Cray Machine, only to install a SISAL compiler that would generate C code for a multiprocessor Cray while pinpointing differences in the executable compiler code. It was thought that these differences would pinpoint where changes should be made in the SISAL Compilers code in order to allow multiprocessing on a SUN network. The applicable sections of the

timer.c file preprocessor directives were commented out in order to obtain complete installation as controlled by the Makefile. This installation appeared to be successful.

a. Where the Reasoning Failed

At this point in the research, it was thought that there was a complete and working SISAL compiler for the SUN workstation located in the /cad.exp/sisal directory and a working SISAL compiler that generated C code for a multiprocessor Cray located in the /cad.exp/sisal/craysisal directory. However, all was not as it appeared. Attempts to test the SISAL Compiler for the Cray were misleading. The same SISAL programs tested above (sum.sis and fact.sis) were compiled, and tested again within the craysisal directory. Or so it was thought.

The SISAL Compiler generates C code. Thus, it was expected that the SUN workstations C compiler would be able to compile the code created by the Cray installation. This was because the SUN's C compiler was identified as the compiler utilized during installation with the Makefile. The programs above were compiled as expected, and were then executed.

The results obtained were identical to those achieved during the previous tests with the SISAL compiler installed for the SUN workstation. It appeared that a successful SISAL Compiler had been created that would generate parallel code for a SUN, by installing the SISAL Compiler as if it were being installed on a Cray.

Investigation into the executable code produced by both compilers showed that the files created in both cases were identical. When comparing the executable code for the compilers, there were differences, as would be expected. Yet, the code produced by the different compilers was identical. How could this be?

When the original SISAL compiler was installed, the path statement in the system startup file was modified to inform the workstation that there were executable programs located in the cad.exp/sisal/bin subdirectory. When the Makefile creates the compiler code, all of the modules (IF1LD, IF2UP, CGEN, etc.) for the compiler are placed in this subdirectory, as well as a file named OSC. It is this file that is called to compile the SISAL program files. Although it was thought that when working in the craysisal subdirectory the programs were compiled with the generated Cray SISAL compiler; this was not the case. The path statements were never modified, and when OSC was called from within the cray SISAL subdirectory, the original OSC compiler, and its associated libraries were used to compile the programs. Thus, the compiled executable files would be identical.

At this point it was determined that installing the SISAL compiler on a SUN that generates C code for a multiprocessor Cray would not work. The path statements were modified. The file OSC in the craysisal/bin subdirectory was renamed OSC1. It was thus assured that this file, and the associated libraries and module files, would be called with the execution of the SISAL compiler to

generate C code for a Cray. Additional attempts to compile and execute the SISAL programs using the compiler for the multiprocessor Cray failed. Although it appeared that the Makefile installed the OSC correctly, the compiler produced could not successfully compile and create executable entities from SISAL programs.

It was determined that the SISAL compiler to produce Cray code installed above was unsatisfactory. Although the Makefile gave the appearance of producing an executable SISAL compiler, the OSC created was unsatisfactory. The changes made in the creation of the module files caused them to produce code which was error prone. Thus, it was determined that another path must be taken in the research.

3. SISAL for a Multiple Processor Sun Workstation

All of the files used above to create a SISAL compiler that would generate C code for a multiproccessor Cray were deleted. As this thought process led to errors in reasoning, and faulty code production, all references were removed so as not to interfere with future work.

Delving deeper into the code used to build the compiler, it was determined that the probable areas to investigate would be in the IF2PART and CGEN modules. These are likely candidates for producing differences in code for multiple processors, as opposed to single processors. However, no indications of what differences would be generated were found. In the file osc.c there is a reference to the variable MAX_PROCS which is defined as a C preprocessor

directive during the execution of the Makefile. This variable is passed as an argument to the associated module files for their compilation and building. There is no reference to this variable in either of these two module files.

During the development of the SISAL language, it was proven to be multiprocessor capable by operating on the Manchester dataflow machine, various Crays, etc. Thus, there is sufficient evidence to conclude that there are some differences in the code produced for any multiprocessor machine and the code produced for a single processor SUN workstation.

It was determined that it may be possible to install the OSC on a SUN workstation with an added twist. The Makefile defines the variable PROCS, the number of available processors on the host system. It uses this variable to define the C preprocessor directive -DMAX_PROCS, which is passed through each phase of the compilers generation and installation. By modifying the Makefile generated for SUN workstations, would it be possible to fool the compiler into thinking that it had multiple processors to work with? The answer is yes.

In the Makefile, the variable PROCS and the preprocessor directive defining MAX_PROCS were changed from 1 to 2. This Makefile (Appendix A) was then run to install a second OSC to determine if any differences would arise. These files, and the associated compiler, are located in the cad.exp/sisal/sisal directory.

a. Testing

All appearances during the installation indicated that once again a successful compiler was created. The appropriate changes to the path statements were made. The file OSC in the /bin subdirectory was renamed OSC1. This ensured that the proper files and libraries would be accessed during SISAL program compilation.

The files sum.sis and fact.sis were again compiled. There were no compilation errors, and the executable code produced the expected results when run. It appears that a multiple processor SISAL compiler for SUN workstations was in the making.

One of the runtime capabilities of SISAL is to provide the programmer with runtime information for their program. The runtime system keeps track of the number of workers, storage allocation, CPUTime, WallTime, etc., during program execution, and will provide this information to the programmer upon completion of execution, if desired. An interesting point is that the compiler developed for two processors actually attempted to keep track of the information on both processors, even though only one processor was present.

This information lead the author to believe that yes, it was possible for the compiler to be installed for multiple processors on a SUN network. Although there is only one processor available, at least the code generated by the compiler will support multiple processors.

4. Source Code Investigation

With this new information in hand, is there a way to exploit the parallelism? Could the code provided to generate the SISAL source compiler be modified to utilize multiple processors? Are there facilities available from the host operating system to aid in the achievement of the desired goal? These questions started to arise when the above results were observed.

While looking through all of the files used to generate the compiler, a file named p-ppp.c (Appendix C) was located in the Runtime subdirectory. This file is compiled, and placed in the library archives for the compiler to access while compiling SISAL programs. Using the UNIX diff command, the p-ppp.o file created for single processor SISAL was compared with the p-ppp.o file created for SISAL with two processors available. Although these object files show no differences, it was thought that this file would eventually come into play.

a. The Pi Program

The documentation downloaded with the SISAL programs is very rich and complete. Every reference provided has a rich set of example programs to be used as a basis for learning the language. These files would also provide a strong functional data base for any SISAL installation.

One of the programs found was a parallel version to calculate pi [Ref. 2]. This program (Appendix B) was written as ppie.sis into the Examples/Sum/pie subdirectory for the multi-processor SISAL. The goal here

was to have a program available in the event that parallel processing abilities were achieved.

Looking deeper into the p-ppp.c library file, it was noted that different code was selected depending upon the host machine. It was also apparent that this was where the multiple processors were assigned the threads from the idle worker pool. Hence, the number of workers is dependent upon the number of processors assigned, which is as would be expected.

The section of code specific to SUN workstations would only handle a single processor, as would be expected. Would modifying this code in any way expand the compilers capabilities on the SUN network? Is this where the hardware parallelism takes place? From the preceding discussion on the SISAL language, it is known that the parallelization of the code occurs in IF2PART. It is also known that parallelized functional languages will operate on sequential machines. However, where does the hardware interface occur? What is the controlling factor for allowing machine application of parallelized code?

While the original SUN SISAL was for a single processor, and the Makefile created specified this, recall that a multiple processor capable OSC had been installed. Looking at the code for p-ppp.c it was noted that the fork process had been utilized for the ENCORE host. Recalling that the fork process is also available in UNIX, could this information shed any new material on the study?

The source code for p-ppp.c was edited and recompiled to implement this change. The inclusion of the fork process was as defined in the ENCORE

specific code with cut and paste. The object code compiled successfully, and replaced the original object code in the library archive. This code was then tested on the parallel version of the pi program discussed above. Although the fork command did not function as expected, it was noted that the inclusion of the suspected parallel code did cause a change in the observed operational parameters in that the programs operational efficiency dropped from near 100% to nearly 50%. In other words, the actual cpu time used by the single processor available was half the wall time required for program execution, (i.e., the wall time increased), whereas for single processor execution the cpu time and wall time were nearly identical. Does this indicate that the inclusion of an additional processor would have improved the efficiency while shortening the total real time required for execution? Would both processors have run to achieve the parallelism desired?

5. Code Comparison

Now that a working copy of SISAL for one and two processors was available, it was necessary to determine where the two codes differ. The goal is to exploit the multiple processors found in the various SUN workstations connected together in a loosely coupled distributed network. It appeared that the best method of determining what changes were necessary would be to compare the codes of the two generated compilers.

The Makefiles used to create the compilers were identical with the exception of the defined variable PROCS, and the preprocessor directive

MAX_PROCS. The directory structure for both compilers are identical. The UNIX diff command was used to compare all files and subdirectories in both directory trees. The only differences found were in the executable file OSC, the object files p-dsa.o, p-init.o, p-interface.o, and the archived library libsisal.a. The source code p-dsa.c, p-init.c, p-interface.c are included in Appendix C. The differences between the two libsisal.a files is attributed to the fact that they are composed by archiving the Runtime object files, of which p-dsa.o, p-init.o and p-interface.o are a part. The differences between the executable OSC files as well as the object files is due to the change in the preprocessor directive MAX_PROCS, which is passed as an argument during the compilation of each of these files. No other differences were found in any of the other files located in the directory structure of both compilers.

6. Program Execution

This phase of the research provided the most insight into what was actually occurring during the compilation of a SISAL Program, as well as indicating the next step. It was achieved by tracing the execution of the executable entity produced by the compilation of ppie.sis using the multiprocessor OSC. The source codes for the library files were modified to insert comments in strategic locations, thereby enabling the tracing of the program.

Each time OSC is installed, be it for a single processor, or for multiple processors, the runtime object file p-srt0.o (Appendix C) is created by compiling p-srt0.c. This file is placed in the /bin directory instead of archived in the library

archives. It is through this file that execution of the executable entity produced by compiling SISAL programs occurs. This object file ensures that the execution of any SISAL Program occurs in the same fashion, independent of the host machine.

Execution commences with the parsing of the command line to obtain any runtime arguments (such as no data collection, no parallel execution, number of workers to utilize, etc.). These arguments are then used to initialize the runtime system. This includes setting up the dynamic storage allocation capabilities of the runtime system, as well as establishing worker assignments. Additionally, if runtime statistics are desired, the data collection facilities to create these statistics are initialized.

Other runtime arguments are passed to SisalMain, where the execution of the compiled SISAL program code is initiated. SisalMain contains the function main, where execution originates. It is this function that actually calls the function main within the SISAL Program, which has been renamed during the creation of the executable entity. It is at this point that the programs slices are assigned to the individual workers to allow execution.

As was expected, the initialization of the worker(s) occurs with the execution of the object code p-ppp.o, which is located in the archived file libsisal.a. It is this code that requires further modification in order to permit the parallel execution of SISAL Programs on the SUN network.

VI. CONCLUSIONS AND RECOMMENDATIONS

A. CONCLUSIONS

It has been demonstrated that the SISAL Language is operable on a stand-alone SUN Workstation. It has also been demonstrated that a compiler for SUN Workstations capable of utilizing two processors can be created, and installed on a SUN Workstation. It is the opinion of the author that a SISAL compiler can be produced that will operate on the distributed network of SUN Workstations.

B. RECOMMENDATIONS

As indicated at the end of the last chapter, the object code p-ppp.o initializes and controls the workers utilized by the Optimizing SISAL Compiler. This file is produced during the installation of the OSC, with the compilation of the file p-ppp.c (Appendix C) which is located in the associated Runtime subdirectory.

This file generates different object code for the library, dependent upon the host machine installed for. With no modification to the existing code for p-ppp.c, when SISAL is compiled on a SUN workstation, it will default to a single processor, no matter what the arguments passed during the installation. This occurs by defining and initializing the variable p_procnum to zero at the beginning of the SUN section of code.

Hence, as the execution of the object file continues to proceed, the dynamic storage allocated is only based on the use of a single processor. Also, when the

worker pool is developed, and loop slices assigned, only one processor is available for execution.

As was discussed in the previous chapter, the author attempted to modify this code to utilize the fork command for process execution. However, not enough research was directed along these lines to determine whether or not this would provide a viable vehicle for obtaining the desired objective. It may be possible that the use of the fork process, combined with the pipe command for interprocess communication and the exec family provided by the UNIX Operating System, will provide the necessary facilities to achieve multiprocessing capabilities.

APPENDIX A
SISAL CONFIGURATION AND INSTALLATION FILES

```
#! /bin/sh

echo "* This script will ask some questions about your system and build a"
echo "* Makefile for osc (Optimizing Sisal Compiler) installation."
echo "* If you already have a file called \"Makefile\" it will be overwritten!"
echo "* For some questions, a default response is given in []."
echo "* Pressing RETURN in response to such a question will enable the default."
echo "* Answer yes/no questions with y or n."
echo ""
echo "* Press RETURN to continue."
read ans

echo " "
echo "Is this system:"
echo "1.  Sun running SunOS"
echo "2.  Some other sequential machine running UNIX"
echo "3.  Sequent Balance running DYNIX"
echo "4.  Alliant FX series running Concentrix"
echo "5.  Encore Multimax running Umax"
echo "6.  Sequent Symmetry running DYNIX"
echo "7.  Cray Y-MP or X-MP running UNICOS"
echo "8.  Cray 2 running UNICOS"
echo "9.  SGI running IRIX"
echo "10. IBM RS6000 running AIX"
echo "Enter a number: [1]"
read ans

LIBM='other'

FPO=''
DPO=''
PAR=''
GANGD=''

TheFF='f77'
TheCC='cc'
TheAR='ar r'
TheINSTALL='/bin/cp'

PROCS='1'

RHOST='-DMY_UGH'
HOST2=''

case "X$ans" in
X|X1) VERSION='sun3';
   HOST='-DSUN';
   echo "Is a 68881 floating point chip installed?";
   read ans;
   if [ $ans = 'y' ]
   then
     FPO='-f68881'
     DPO='-DF68881'
   fi ;;
```

```
X2) VERSION='sunix';
   HOST='-DSUNIX' ;;
X3) VERSION='balance';
   HOST='-DBALANCE';
   echo "Enter the number of available processors";
   read ans;
   PROCS=$ans;
   TheFF='fortran'
   TheCC='cc -i';
   echo "Run make in parallel?";
   read ans;
   if [ $ans = 'y' ]
   then
    PAR='&'
   fi;
   echo "Is LLNL gang daemon software installed?";
   read ans;
   if [ $ans = 'y' ]
   then
    GANGD='-DGANGD'
   fi ;;
X4) VERSION='alliant';
   HOST='-DALLIANT';
   TheFF='fortran'
   TheCC='fxc -w -ce -OM';
   LIBM="alliant";
   echo "Enter the number of available processors";
   read ans;
   PROCS=$ans ;;
X5) VERSION='encore';
   HOST='-DENCORE';
   echo "Enter the number of available processors";
   read ans;
   PROCS=$ans ;;
X6) VERSION='symmetry';
   HOST='-DSYMMETRY';
   TheFF='fortran'
   TheCC='cc -i'
   echo "Enter the number of available processors";
   read ans;
   PROCS=$ans;
   echo "Run make in parallel?";
   read ans;
   if [ $ans = 'y' ]
   then
    PAR='&'
   fi;
   echo "Are Weitek 1167 floating point accelerators installed?";
   read ans;
   if [ $ans = 'y' ]
   then
    FPO='-f1167'
    DPO='-Dw1167'
   fi ;;
```

```
X7) VERSION='cray';
   HOST='-DCRAY';
   RHOST='-DCRAYXY';
   TheCC='scc';
   TheFF='cf77'
   LIBM='cray';
   TheAR='bld r'
   echo "Enter the number of available processors";
   read ans;
   PROCS=$ans;;
X8) VERSION='cray';
   HOST='-DCRAY';
   RHOST='-DCRAY2';
   TheCC='scc';
   TheFF='cf77'
   LIBM='cray';
   TheAR='bld r'
   echo "Enter the number of available processors";
   read ans;
   PROCS=$ans;;
X9) VERSION='sgi';
   LIBM="sgi";
   HOST='-DSGI';
   echo ""
   echo "* WARNING: The Sisal run time system on the SGI uses schedctl to"
   echo "*        establish gang management of parallel execution and "
   echo "*        sysmp to bind the worker processes (Runtime/p-ppp.c)."
   echo ""
   echo "Enter the number of available processors";
   read ans;
   PROCS=$ans;;
X10) VERSION='rs6000';
   HOST='-DRS6000' ;;
*)  VERSION='unknown';
   HOST='-DUNKNOWN' ;;
esac

if [ $RHOST = '-DCRAYXY' ]
then
   echo ""
   echo "* WARNING: The Sisal run time system on the CRAY X-MP and Y-MP"
   echo "*        uses _semclr(31) and _semts(31) as well as LOCKTEST,"
   echo "*        LOCKON, and LOCKOFF, assuming that the microtasking"
   echo "*        library will not interfere with the 2 intrinsics. If"
   echo "*        this is not the case, then make approapriate changes"
   echo "*        to Runtime/locks.h."
   echo ""
fi

if [ $RHOST = '-DCRAY2' ]
then
   echo ""
   echo "* WARNING: The Sisal run time system on the CRAY 2 uses _getsem"
   echo "*        and _csem as well as LOCKTEST, LOCKON, and LOCKOFF,"
```

```
   echo "*        assuming that the microtasking library will not "
   echo "*        interfere with the 2 intrinsics.  If this is not the"
   echo "*        case, then make appropriate changes to Runtime/locks.h."
   echo ""
fi

if [ $VERSION = 'cray' ]
then
   TheRANLIB='touch'
   TheTIMEM=''
else
   TheRANLIB='touch'
   echo "Is ranlib supported? [y]"
   read ans
   case "X$ans" in
    X|Xy) TheRANLIB='ranlib'
   esac

   TheTIMEM='-DUSE_TIMES'
   echo "Is getrusage supported? [y]"
   read ans
   case "X$ans" in
    X|Xy) TheTIMEM=";;
    Xn) echo ""
       echo "* WARNING: The Sisal run time system will use TIMES to gather"
       echo "*        timing information during execution. On the SGI the"
       echo "*        HZ value is assumed to be 100. On all other machines"
       echo "*        it is assumed to be 60 (man times,Runtime/p-timer.c,"
       echo "*        Backend/If1opt/if1timer.c)."
       echo "";
       esac
fi;

OPT=''
ROPT=''
echo "Optimize the installed code? [y]"
read ans
case "X$ans" in
  X|Xy) OPT='-O';
  ROPT='-O'
  if [ $HOST = '-DCRAY' ]
  then ROPT=''
    OPT=''
 fi
 if [ $HOST = '-DALLIANT' ]
 then ROPT='-Oigv'
     OPT='-Oig'
 fi;;
  Xn)   if [ $HOST = '-DCRAY' ]
 then ROPT='-h noopt'
    OPT='-h noopt'
       fi;;
esac
```

```
DBX='-g'
echo "Compile for run time dbx use via \"-g\"? [n]"
read ans
case "X$ans" in
  X|Xn) DBX=''
esac

echo "Enter path to directory for executables: [/usr/local/bin]"
read BIN_PATH
if [ "X$BIN_PATH" = 'X' ]
then
  BIN_PATH='/usr/local/bin'
fi

echo "Enter path to man pages: [/usr/man/man1]"
read MAN_PATH
if [ "X$MAN_PATH" = 'X' ]
then
  MAN_PATH='/usr/man/man1'
fi

echo ""
echo "* Makefile construction in progress..."
DATE=`date`
cat >Makefile <<EOF
# Makefile for SISAL
# Generated $DATE by $0.

# ********************************************************************************
# ************************  MACROS TO CONFIGURE MAKEFILE  ************************
# ********************************************************************************
# **** DOCUMENTATION SYMOBLS: [] = optional, {} = pick one
# **** COMMAND LINE MACRO DEFINITIONS WILL OVERRIDE THOSE SHOWN HERE

# **** PATHS TO COMMANDS USED BY THE MAKEFILE--CHECK FOR ACCURACY
CC        = $TheCC
INSTALL   = $TheINSTALL
RANLIB    = $TheRANLIB
AR        = $TheAR

# **** HOST SYSTEM
# **** HOST = -D{ENCORE,ALLIANT,etc.}
HOST = $HOST $RHOST

# *** NUMBER OF AVAILABLE PROCESSORS IN THE HOST SYSTEM
PROCS = $PROCS

# **** FLOATING POINT CHIP (FOR EXAMPLE, SUN)
# **** FPO = -f68881
FPO = $FPO
DPO = $DPO

# **** IS GANGD TO BE USED: ONLY SUPPORTED ON BALANCE
# **** GANGD = [-DGANGD]
```

```
GANGD = $GANGD

# **** SHOULD THE MAKEFILE GO IN PARALLEL: ONLY SUPPORTED ON BALANCE
# **** PAR = [&]
PAR = $PAR

# **** OPTIMIZE THE GENERATED ASSEMBLY CODE
# **** OPT = [-O]
OPT = $OPT
ROPT = $ROPT

# **** COMPILE FOR RUN TIME DBX USE
# **** DBX = [-g]
DBX = $DBX

# **** ABSOLUTE PATHS TO EXECUTABLE (BIN_PATH) AND MAN PAGE INSTALLATION
# **** SITES. (Ex. BIN_PATH = /usr/local/bin, MAN_PATH = /usr/man/manl)
BIN_PATH = $BIN_PATH
MAN_PATH = $MAN_PATH


#*****************************************************************************
# ************************ DO NOT MODIFY ANYTHING ELSE ************************
#*****************************************************************************

PR      = -DMAX_PROCS=${PROCS}
FF      = ${TheFF}

T_DEF1    = BIN_PATH=\${BIN_PATH}
T_DEF2    = MAN_PATH=\${MAN_PATH}

LIBM      = ${LIBM}

TIMEM     = ${TheTIMEM}

CC_OPTS   = \${HOST} \${PR} \${FPO} \${DPO} \${GANGD} \${DBX} \${TIMEM}


F_CC_CMD  = "CC=\${CC} \${HOST}"
T_CC_CMD  = "CC=\${CC} \${CC_OPTS} \${OPT}" "\${T_DEF1}" "\${T_DEF2}" \
    "PAR=\${PAR}"
R_CC_CMD  = "CC=\${CC} \${CC_OPTS} \${ROPT}" "PAR=\${PAR}" "AR=\${AR}"
CC_CMD    = "CC=\${CC} \${CC_OPTS} \${OPT}" "PAR=\${PAR}" "AR=\${AR}"


BINSTALL  = \${INSTALL}
SINSTALL  = \${INSTALL}

BINSTALL_CMD = "BINSTALL=\${BINSTALL}" "BIN_PATH=\${BIN_PATH}"
SINSTALL_CMD = "SINSTALL=\${SINSTALL}" "MAN_PATH=\${MAN_PATH}"


# **** LOCAL (all without install)
local: tools frontend backend runtime
```

```
# **** ALL
all: tools frontend backend runtime install

# **** FRONTEND
frontend:
cd Frontend; make local \${F_CC_CMD}; cd ..

# **** TOOLS
tools:
cd Tools; make local \${T_CC_CMD}; cd ..

# **** BACKEND
backend: if1ld if1opt if2mem if2up if2part if2gen
if1ld:
cd Backend/If1ld;    make local \${CC_CMD}; cd ../..
if1opt:
cd Backend/If1opt;   make local \${CC_CMD}; cd ../..
if2mem:
cd Backend/If2mem;   make local \${CC_CMD}; cd ../..
if2up:
cd Backend/If2up;    make local \${CC_CMD}; cd ../..
if2part:
cd Backend/If2part;  make local \${CC_CMD}; cd ../..
if2gen:
cd Backend/If2gen;  make local \${CC_CMD}; cd ../..

# **** RUNTIME
runtime:
cd Runtime; make local \${R_CC_CMD}; cd ..

# **** INSTALL
install: install_tools install_if1ld install_if1opt install_if2mem \\
 install_if2up install_if2part install_if2gen install_runtime \\
 install_frontend

install_tools:
cd Tools; \\
make install -i \${T_CC_CMD} \${BINSTALL_CMD} \${SINSTALL_CMD};\\
cd ..
install_frontend:
cd Frontend;   make install -i \${BINSTALL_CMD} \${SINSTALL_CMD};\\
cd ..
install_if1ld:
cd Backend/If1ld;   make install -i \${BINSTALL_CMD} \${SINSTALL_CMD};\\
cd ../..
install_if1opt:
cd Backend/If1opt;  make install -i \${BINSTALL_CMD} \${SINSTALL_CMD};\\
cd ../..
install_if2mem:
cd Backend/If2mem;  make install -i \${BINSTALL_CMD} \${SINSTALL_CMD};\\
cd ../..
install_if2up:
cd Backend/If2up;   make install -i \${BINSTALL_CMD} \${SINSTALL_CMD};\\
```

```
cd ../..
install_if2part:
cd Backend/If2part; make install -i \${BINSTALL_CMD} \${SINSTALL_CMD};\\
cd ../..
install_if2gen:
cd Backend/If2gen; make install -i \${BINSTALL_CMD} \${SINSTALL_CMD};\\
cd ../..
install_runtime:
cd Runtime;\\
make \${LIBM} install -i \${BINSTALL_CMD} \${SINSTALL_CMD} "FF=\${FF}" "AR=\${AR}" "RANLI-
B=\${RANLIB}";\\
cd ../..

# **** CLEAN
clean:
cd Tools;           make clean; cd ..
cd Frontend;        make clean; cd ..
cd Backend/If1ld;   make clean; cd ../..
cd Backend/If1opt;  make clean; cd ../..
cd Backend/If2mem;  make clean; cd ../..
cd Backend/If2up;   make clean; cd ../..
cd Backend/If2part; make clean; cd ../..
cd Backend/If2gen;  make clean; cd ../..
cd Runtime;         make clean; cd ..
EOF

echo ""
echo "* Makefile has been built."
echo "* Please check it over to ensure it is as you wish."
echo "* When you are satisfied, enter \"make all\" to build and install osc."
```

```
# Makefile for SISAL
# Generated Thu Sep  3 16:44:52 PDT 1992 by sinstall.

# ****************************************************************************
# ************************  MACROS TO CONFIGURE MAKEFILE  ********************
# ****************************************************************************
# **** DOCUMENTATION SYMOBLS: [] = optional, { } = pick one
# **** COMMAND LINE MACRO DEFINITIONS WILL OVERRIDE THOSE SHOWN HERE

# **** PATHS TO COMMANDS USED BY THE MAKEFILE--CHECK FOR ACCURACY
CC        = cc
INSTALL   = /bin/cp
RANLIB    = ranlib
AR        = ar r

# **** HOST SYSTEM
# **** HOST = -D{ENCORE,ALLIANT,etc.}
HOST = -DSUN -DMY_UGH

# *** NUMBER OF AVAILABLE PROCESSORS IN THE HOST SYSTEM
PROCS = 1

# **** FLOATING POINT CHIP (FOR EXAMPLE, SUN)
# **** FPO = -f68881
FPO =
DPO =

# **** IS GANGD TO BE USED: ONLY SUPPORTED ON BALANCE
# **** GANGD = [-DGANGD]
GANGD =

# **** SHOULD THE MAKEFILE GO IN PARALLEL: ONLY SUPPORTED ON BALANCE
# **** PAR = [&]
PAR =

# **** OPTIMIZE THE GENERATED ASSEMBLY CODE
# **** OPT = [-O]
OPT = -O
ROPT = -O

# **** COMPILE FOR RUN TIME DBX USE
# **** DBX = [-g]
DBX =

# **** ABSOLUTE PATHS TO EXECUTABLE (BIN_PATH) AND MAN PAGE INSTALLATION
# **** SITES. (Ex. BIN_PATH = /usr/local/bin, MAN_PATH = /usr/man/manl)
BIN_PATH = /cad.exp/sisal/bin
MAN_PATH = /cad.exp/sisal/man


#*****************************************************************************
# ************************ DO NOT MODIFY ANYTHING ELSE ************************
#*****************************************************************************
```

```
PR       = -DMAX_PROCS=1
FF       = f77

T_DEF1   = BIN_PATH=${BIN_PATH}
T_DEF2   = MAN_PATH=${MAN_PATH}

LIBM     = other

TIMEM    =

CC_OPTS  = ${HOST} ${PR} ${FPO} ${DPO} ${GANGD} ${DBX} ${TIMEM}


F_CC_CMD  = "CC=${CC} ${HOST}"
T_CC_CMD  = "CC=${CC} ${CC_OPTS} ${OPT}" "${T_DEF1}" "${T_DEF2}"     "PAR=${PAR}"
R_CC_CMD  = "CC=${CC} ${CC_OPTS} ${ROPT}" "PAR=${PAR}" "AR=${AR}"
CC_CMD    = "CC=${CC} ${CC_OPTS} ${OPT}" "PAR=${PAR}" "AR=${AR}"


BINSTALL  = ${INSTALL}
SINSTALL  = ${INSTALL}

BINSTALL_CMD = "BINSTALL=${BINSTALL}" "BIN_PATH=${BIN_PATH}"
SINSTALL_CMD = "SINSTALL=${SINSTALL}" "MAN_PATH=${MAN_PATH}"


# **** LOCAL (all without install)
local: tools frontend backend runtime

# **** ALL
all: tools frontend backend runtime install

# **** FRONTEND
frontend:
cd Frontend; make local ${F_CC_CMD}; cd ..

# **** TOOLS
tools:
cd Tools; make local ${T_CC_CMD}; cd ..

# **** BACKEND
backend: if1ld if1opt if2mem if2up if2part if2gen
if1ld:
cd Backend/If1ld;   make local ${CC_CMD}; cd ../..
if1opt:
cd Backend/If1opt;  make local ${CC_CMD}; cd ../..
if2mem:
cd Backend/If2mem;  make local ${CC_CMD}; cd ../..
if2up:
cd Backend/If2up;   make local ${CC_CMD}; cd ../..
if2part:
cd Backend/If2part; make local ${CC_CMD}; cd ../..
if2gen:
cd Backend/If2gen;  make local ${CC_CMD}; cd ../..
```

```
# **** RUNTIME
runtime:
cd Runtime; make local ${R_CC_CMD}; cd ..

# **** INSTALL
install: install_tools install_if1ld install_if1opt install_if2mem \
 install_if2up install_if2part install_if2gen install_runtime \
 install_frontend

install_tools:
cd Tools; \
make install -i ${T_CC_CMD} ${BINSTALL_CMD} ${SINSTALL_CMD};\
cd ..
install_frontend:
cd Frontend;   make install -i ${BINSTALL_CMD} ${SINSTALL_CMD};\
cd ..
install_if1ld:
cd Backend/If1ld;   make install -i ${BINSTALL_CMD} ${SINSTALL_CMD};\
cd ../..
install_if1opt:
cd Backend/If1opt;  make install -i ${BINSTALL_CMD} ${SINSTALL_CMD};\
cd ../..
install_if2mem:
cd Backend/If2mem;  make install -i ${BINSTALL_CMD} ${SINSTALL_CMD};\
cd ../..
install_if2up:
cd Backend/If2up;   make install -i ${BINSTALL_CMD} ${SINSTALL_CMD};\
cd ../..
install_if2part:
cd Backend/If2part; make install -i ${BINSTALL_CMD} ${SINSTALL_CMD};\
cd ../..
install_if2gen:
cd Backend/If2gen; make install -i ${BINSTALL_CMD} ${SINSTALL_CMD};\
cd ../..
install_runtime:
cd Runtime;\
make ${LIBM} install -i ${BINSTALL_CMD} ${SINSTALL_CMD} "FF=${FF}"
      "AR=${AR}" "RANLIB=${RANLIB}";\
cd ../..

# **** CLEAN
clean:
cd Tools;          make clean; cd ..
cd Frontend;       make clean; cd ..
cd Backend/If1ld;   make clean; cd ../..
cd Backend/If1opt;  make clean; cd ../..
cd Backend/If2mem;  make clean; cd ../..
cd Backend/If2up;   make clean; cd ../..
cd Backend/If2part; make clean; cd ../..
cd Backend/If2gen;  make clean; cd ../..
cd Runtime;        make clean; cd ..
```

```
# Makefile for SISAL
# Generated Thu Feb 25 22:30:16 PST 1993 by sinstall.

# ****************************************************************************
# ************************  MACROS TO CONFIGURE MAKEFILE  ********************
# ****************************************************************************
# **** DOCUMENTATION SYMOBLS: [] = optional, { } = pick one
# **** COMMAND LINE MACRO DEFINITIONS WILL OVERRIDE THOSE SHOWN HERE

# **** PATHS TO COMMANDS USED BY THE MAKEFILE--CHECK FOR ACCURACY
CC        = cc
INSTALL   = /bin/cp
RANLIB    = touch
AR        = ar r

# **** HOST SYSTEM
# **** HOST = -D{ENCORE,ALLIANT,etc.}
HOST = -DCRAY -DCRAYXY

# *** NUMBER OF AVAILABLE PROCESSORS IN THE HOST SYSTEM
PROCS = 2

# **** FLOATING POINT CHIP (FOR EXAMPLE, SUN)
# **** FPO = -f68881
FPO =
DPO =

# **** IS GANGD TO BE USED: ONLY SUPPORTED ON BALANCE
# **** GANGD = [-DGANGD]
GANGD =

# **** SHOULD THE MAKEFILE GO IN PARALLEL: ONLY SUPPORTED ON BALANCE
# **** PAR = [&]
PAR =

# **** OPTIMIZE THE GENERATED ASSEMBLY CODE
# **** OPT = [-O]
OPT =
ROPT =

# **** COMPILE FOR RUN TIME DBX USE
# **** DBX = [-g]
DBX =

# **** ABSOLUTE PATHS TO EXECUTABLE (BIN_PATH) AND MAN PAGE INSTALLATION
# **** SITES. (Ex. BIN_PATH = /usr/local/bin, MAN_PATH = /usr/man/manl)
BIN_PATH = /cad.exp/sisal/craysisal/bin
MAN_PATH = /cad.exp/sisal/craysisal/Man/manl


#*****************************************************************************
# ************************ DO NOT MODIFY ANYTHING ELSE ************************
#*****************************************************************************
```

```
PR      = -DMAX_PROCS=2
FF      = f77

T_DEF1    = BIN_PATH=${BIN_PATH}
T_DEF2    = MAN_PATH=${MAN_PATH}

LIBM      = other

TIMEM     =

CC_OPTS   = ${HOST} ${PR} ${FPO} ${DPO} ${GANGD} ${DBX} ${TIMEM}


F_CC_CMD  = "CC=${CC} ${HOST}"
T_CC_CMD  = "CC=${CC} ${CC_OPTS} ${OPT}" "${T_DEF1}" "${T_DEF2}"      "PAR=${PAR}"
R_CC_CMD  = "CC=${CC} ${CC_OPTS} ${ROPT}" "PAR=${PAR}" "AR=${AR}"
CC_CMD    = "CC=${CC} ${CC_OPTS} ${OPT}" "PAR=${PAR}" "AR=${AR}"


BINSTALL  = ${INSTALL}
SINSTALL  = ${INSTALL}

BINSTALL_CMD = "BINSTALL=${BINSTALL}" "BIN_PATH=${BIN_PATH}"
SINSTALL_CMD = "SINSTALL=${SINSTALL}" "MAN_PATH=${MAN_PATH}"


# **** LOCAL (all without install)
local: tools frontend backend runtime

# **** ALL
all: tools frontend backend runtime install

# **** FRONTEND
frontend:
cd Frontend; make local ${F_CC_CMD}; cd ..

# **** TOOLS
tools:
cd Tools; make local ${T_CC_CMD}; cd ..

# **** BACKEND
backend: if1ld if1opt if2mem if2up if2part if2gen
if1ld:
cd Backend/If1ld;   make local ${CC_CMD}; cd ../..
if1opt:
cd Backend/If1opt;  make local ${CC_CMD}; cd ../..
if2mem:
cd Backend/If2mem;  make local ${CC_CMD}; cd ../..
if2up:
cd Backend/If2up;   make local ${CC_CMD}; cd ../..
if2part:
cd Backend/If2part; make local ${CC_CMD}; cd ../..
if2gen:
cd Backend/If2gen;  make local ${CC_CMD}; cd ../..
```

```
# **** RUNTIME
runtime:
cd Runtime; make local ${R_CC_CMD}; cd ..

# **** INSTALL
install: install_tools install_if1ld install_if1opt install_if2mem \
 install_if2up install_if2part install_if2gen install_runtime \
 install_frontend

install_tools:
cd Tools; \
make install -i ${T_CC_CMD} ${BINSTALL_CMD} ${SINSTALL_CMD};\
cd ..
install_frontend:
cd Frontend;   make install -i ${BINSTALL_CMD} ${SINSTALL_CMD};\
cd ..
install_if1ld:
cd Backend/If1ld;   make install -i ${BINSTALL_CMD} ${SINSTALL_CMD};\
cd ../..
install_if1opt:
cd Backend/If1opt;  make install -i ${BINSTALL_CMD} ${SINSTALL_CMD};\
cd ../..
install_if2mem:
cd Backend/If2mem;  make install -i ${BINSTALL_CMD} ${SINSTALL_CMD};\
cd ../..
install_if2up:
cd Backend/If2up;   make install -i ${BINSTALL_CMD} ${SINSTALL_CMD};\
cd ../..
install_if2part:
cd Backend/If2part; make install -i ${BINSTALL_CMD} ${SINSTALL_CMD};\
cd ../..
install_if2gen:
cd Backend/If2gen; make install -i ${BINSTALL_CMD} ${SINSTALL_CMD};\
cd ../..
install_runtime:
cd Runtime;\
make ${LIBM} install -i ${BINSTALL_CMD} ${SINSTALL_CMD} "FF=${FF}" "AR=${AR}" "RANLI-
B=${RANLIB}";\
cd ../..

# **** CLEAN
clean:
cd Tools;          make clean; cd ..
cd Frontend;       make clean; cd ..
cd Backend/If1ld;    make clean; cd ../..
cd Backend/If1opt;   make clean; cd ../..
cd Backend/If2mem;   make clean; cd ../..
cd Backend/If2up;    make clean; cd ../..
cd Backend/If2part;  make clean; cd ../..
cd Backend/If2gen;   make clean; cd ../..
cd Runtime;        make clean; cd ..
```

```
# Makefile for SISAL
# Generated Wed Jan 20 11:14:00 PST 1993 by sinstall.

# ****************************************************************************
# ************************  MACROS TO CONFIGURE MAKEFILE  ********************
# ****************************************************************************
# **** DOCUMENTATION SYMOBLS: [] = optional, { } = pick one
# **** COMMAND LINE MACRO DEFINITIONS WILL OVERRIDE THOSE SHOWN HERE

# **** PATHS TO COMMANDS USED BY THE MAKEFILE--CHECK FOR ACCURACY
CC      = cc
INSTALL   = /bin/cp
RANLIB    = ranlib
AR      = ar r

# **** HOST SYSTEM
# **** HOST = -D{ENCORE,ALLIANT,etc.}
HOST = -DSUN -DMY_UGH

# *** NUMBER OF AVAILABLE PROCESSORS IN THE HOST SYSTEM
PROCS = 2

# **** FLOATING POINT CHIP (FOR EXAMPLE, SUN)
# **** FPO = -f68881
FPO =
DPO =

# **** IS GANGD TO BE USED: ONLY SUPPORTED ON BALANCE
# **** GANGD = [-DGANGD]
GANGD =

# **** SHOULD THE MAKEFILE GO IN PARALLEL: ONLY SUPPORTED ON BALANCE
# **** PAR = [&]
PAR =

# **** OPTIMIZE THE GENERATED ASSEMBLY CODE
# **** OPT = [-O]
OPT = -O
ROPT = -O

# **** COMPILE FOR RUN TIME DBX USE
# **** DBX = [-g]
DBX =

# **** ABSOLUTE PATHS TO EXECUTABLE (BIN_PATH) AND MAN PAGE INSTALLATION
# **** SITES. (Ex. BIN_PATH = /usr/local/bin, MAN_PATH = /usr/man/manl)
BIN_PATH = /cad.exp/sisal/sisal/bin
MAN_PATH = /cad.exp/sisal/sisal/man


#*****************************************************************************
# ************************ DO NOT MODIFY ANYTHING ELSE ************************
#*****************************************************************************
```

```
PR      = -DMAX_PROCS=2
FF      = f77

T_DEF1    = BIN_PATH=${BIN_PATH}
T_DEF2    = MAN_PATH=${MAN_PATH}

LIBM      = other

TIMEM     =

CC_OPTS   = ${HOST} ${PR} ${FPO} ${DPO} ${GANGD} ${DBX} ${TIMEM}


F_CC_CMD  = "CC=${CC} ${HOST}"
T_CC_CMD  = "CC=${CC} ${CC_OPTS} ${OPT}" "${T_DEF1}" "${T_DEF2}"      "PAR=${PAR}"
R_CC_CMD  = "CC=${CC} ${CC_OPTS} ${ROPT}" "PAR=${PAR}" "AR=${AR}"
CC_CMD    = "CC=${CC} ${CC_OPTS} ${OPT}" "PAR=${PAR}" "AR=${AR}"


BINSTALL  = ${INSTALL}
SINSTALL  = ${INSTALL}

BINSTALL_CMD = "BINSTALL=${BINSTALL}" "BIN_PATH=${BIN_PATH}"
SINSTALL_CMD = "SINSTALL=${SINSTALL}" "MAN_PATH=${MAN_PATH}"


# **** LOCAL (all without install)
local: tools frontend backend runtime

# **** ALL
all: tools frontend backend runtime install

# **** FRONTEND
frontend:
cd Frontend; make local ${F_CC_CMD}; cd ..

# **** TOOLS
tools:
cd Tools; make local ${T_CC_CMD}; cd ..

# **** BACKEND
backend: if1ld if1opt if2mem if2up if2part if2gen
if1ld:
cd Backend/If1ld;   make local ${CC_CMD}; cd ../..
if1opt:
cd Backend/If1opt;  make local ${CC_CMD}; cd ../..
if2mem:
cd Backend/If2mem;  make local ${CC_CMD}; cd ../..
if2up:
cd Backend/If2up;   make local ${CC_CMD}; cd ../..
if2part:
cd Backend/If2part; make local ${CC_CMD}; cd ../..
if2gen:
cd Backend/If2gen;  make local ${CC_CMD}; cd ../..
```

```
# **** RUNTIME
runtime:
cd Runtime; make local ${R_CC_CMD}; cd ..

# **** INSTALL
install: install_tools install_if1ld install_if1opt install_if2mem \
 install_if2up install_if2part install_if2gen install_runtime \
 install_frontend

install_tools:
cd Tools; \
make install -i ${T_CC_CMD} ${BINSTALL_CMD} ${SINSTALL_CMD};\
cd ..
install_frontend:
cd Frontend;   make install -i ${BINSTALL_CMD} ${SINSTALL_CMD};\
cd ..
install_if1ld:
cd Backend/If1ld;   make install -i ${BINSTALL_CMD} ${SINSTALL_CMD};\
cd ../..
install_if1opt:
cd Backend/If1opt;  make install -i ${BINSTALL_CMD} ${SINSTALL_CMD};\
cd ../..
install_if2mem:
cd Backend/If2mem;  make install -i ${BINSTALL_CMD} ${SINSTALL_CMD};\
cd ../..
install_if2up:
cd Backend/If2up;   make install -i ${BINSTALL_CMD} ${SINSTALL_CMD};\
cd ../..
install_if2part:
cd Backend/If2part; make install -i ${BINSTALL_CMD} ${SINSTALL_CMD};\
cd ../..
install_if2gen:
cd Backend/If2gen; make install -i ${BINSTALL_CMD} ${SINSTALL_CMD};\
cd ../..
install_runtime:
cd Runtime;\
make ${LIBM} install -i ${BINSTALL_CMD} ${SINSTALL_CMD} "FF=${FF}"
      "AR=${AR}" "RANLIB=${RANLIB}";\
cd ../..

# **** CLEAN
clean:
cd Tools;          make clean; cd ..
cd Frontend;       make clean; cd ..
cd Backend/If1ld;   make clean; cd ../..
cd Backend/If1opt;  make clean; cd ../..
cd Backend/If2mem;  make clean; cd ../..
cd Backend/If2up;   make clean; cd ../..
cd Backend/If2part; make clean; cd ../..
cd Backend/If2gen;  make clean; cd ../..
cd Runtime;        make clean; cd ..
```

APPENDIX B
EXAMPLE SISAL PROGRAMS

```
%$entry=example1
define example1

#define VALUE_TYPE double_real
#define VALUE 1.0D0

function example1( n:integer returns VALUE_TYPE )
  for i in 1,n returns value of sum VALUE end for
end function
```

```
define Main

%Main(4) -> 4*3*2*1

function Main( n:integer returns integer)
  if (n <= 0 ) then 1 else n*Main(n-1) end if
end function
```

```
% ************************************************************************
define Main

%
% Main(Cycles) yields an approximation to pi using
%       (4/N)*sum(j=1,N)(1/1+x[j]^2), where x[j] = (j-1/2)/N.
%       See Karp and Babb, "A comparison of 12 parallel FORTRAN
%       dialects," IEEE Software (September 1988):  52-67.
%
% Main(10000) -> 3.141591
%

function Main ( Cycles:integer returns real )
  (4.0/real(Cycles)) * for j in 1,Cycles
                  x := (real(j)-0.5)/real(Cycles)
                returns value of sum 1.0/(1.0+x*x)
                end for
end function
```

APPENDIX C
SISAL RUNTIME FILES

```
#include "world.h"


static char *SharedBase;
static char *SharedMemory;
static int   SharedSize;

#ifndef RS6000
extern char *malloc();
#endif


POINTER SharedMalloc( NumBytes )
int NumBytes;
{
 register char *ReturnPtr;

 NumBytes += 50;
 NumBytes = ALIGN( int, NumBytes );

 if ( SharedSize < NumBytes )
  SisalError( "SharedMalloc", "ALLOCATION SIZE TO BIG" );

 ReturnPtr    = SharedMemory;
 SharedMemory += NumBytes;
 SharedSize   -= NumBytes;

 return( (POINTER) ReturnPtr );
}


#ifdef ENCORE
int p_procnum = 0;
char *share_malloc();

void ReleaseSharedMemory()
{
}

void AcquireSharedMemory( NumBytes )
int NumBytes;
{
 SharedSize = NumBytes + 100000;

 if ( share_malloc_init( SharedSize+100000 ) != 0 )
  SisalError( "AcquireSharedMemory", "share_malloc_init FAILED" );

 SharedBase = SharedMemory = share_malloc( SharedSize-40 );

 if ( SharedMemory == (char *) NULL )
  SisalError( "AcquireSharedMemory", "share_malloc FAILED" );

 SharedMemory = ALIGN(char*,SharedMemory);
}
```

```
void StartWorkers()
{
 register int NumProcs = NumWorkers;

 while( --NumProcs > 0 )
  if ( fork() == 0 )
   break;

 EnterWorker( p_procnum = NumProcs );

 if ( NumProcs != 0 ) {
  LeaveWorker();
  exit( 0 );
  }
}

void StopWorkers()
{
 *SisalShutDown = TRUE;
 LeaveWorker();
}

void AbortParallel()
{
 kill( 0, SIGKILL );
}
#endif



#if SUNIX || SUN || RS6000
int p_procnum = 0;

void ReleaseSharedMemory()
{
 free( SharedBase );
}

void AcquireSharedMemory( NumBytes )
int NumBytes;
{
 SharedSize = NumBytes + 100000;

 SharedBase = SharedMemory = malloc( SharedSize-40 );

 if ( SharedMemory == (char *) NULL )
  SisalError( "AcquireSharedMemory", "malloc FAILED" );

 SharedMemory = ALIGN(char*,SharedMemory);
}

void StartWorkers()
```

```
{
 EnterWorker( p_procnum );
}

void StopWorkers()
{
 *SisalShutDown = TRUE;
 LeaveWorker();
}

void AbortParallel()
{
 exit( 1 );
}
#endif



#ifdef ALLIANT
void ReleaseSharedMemory()
{
 free( SharedBase );
}

void AcquireSharedMemory( NumBytes )
int NumBytes;
{
 SharedSize = NumBytes + 100000;

 SharedBase = SharedMemory = malloc( SharedSize-40 );

 if ( SharedMemory == (char *) NULL )
   SisalError( "AcquireSharedMemory", "malloc FAILED" );

 SharedMemory = ALIGN(char*,SharedMemory);
}

void StartWorkers()
{
 EnterWorker( 0 );
}

void StopWorkers()
{
 *SisalShutDown = TRUE;
 LeaveWorker();
}

void AbortParallel()
{
 kill( 0, SIGKILL );
}
#endif
```

```
#if BALANCE || SYMMETRY
extern char *shmalloc();

void AcquireSharedMemory( NumBytes )
int NumBytes;
{
 SharedSize = NumBytes + 100000;

 SharedBase = SharedMemory = shmalloc( SharedSize-40 );

 if ( SharedMemory == (char *) NULL )
  SisalError( "AcquireSharedMemory", "shmalloc FAILED" );

 SharedMemory = ALIGN(char*,SharedMemory);
}

void ReleaseSharedMemory()
{
 shfree( SharedBase );
}

#ifdef GANGD
void StartWorkers()
{
 register int pID;

 begin_parallel( NumWorkers );

 GETPROCID(pID);

 EnterWorker( pID );

 if ( pID != 0 ) {
  LeaveWorker();
  end_parallel();
  }
}

void StopWorkers()
{
 *SisalShutDown = TRUE;
 LeaveWorker();
 end_parallel();
}

void AbortParallel()
{
 abort_parallel();
}
#else
```

```
int p_procnum = 0;

void StartWorkers( )
{
 register int NumProcs = NumWorkers;

 while( --NumProcs > 0 )
  if ( fork() == 0 )
   break;

 EnterWorker( p_procnum = NumProcs );

 if ( NumProcs != 0 ) {
  LeaveWorker();
  exit( 0 );
  }
}

void StopWorkers()
{
 *SisalShutDown = TRUE;
 LeaveWorker();
}

void AbortParallel()
{
 kill( 0, SIGKILL );
}
#endif

#endif



#ifdef CRAY
int TaskInfo[10][3];
LOCK_TYPE TheFirstLock;

void ReleaseSharedMemory()
{
 free( SharedBase );
}

void AcquireSharedMemory( NumBytes )
int NumBytes;
{
 SharedSize = NumBytes + 100000;

 SharedBase = SharedMemory = malloc( SharedSize-40 );

 if ( SharedMemory == (char *) NULL )
  SisalError( "AcquireSharedMemory", "malloc FAILED" );
```

```
 SharedMemory = ALIGN(char*,SharedMemory);
}

int ProcessorId()
{
 register int pID;
 GETPROCID(pID);
 return( pID );
}

static void CrayWorker( ProcId )
int ProcId;
{
 EnterWorker( ProcId );
 LeaveWorker();
}

void StartWorkers()
{
 register int NumProcs = NumWorkers;
 register int i;

 MY_LOCKASGN;

 for ( i = 0; i < NumProcs; i++ ) {
  TaskInfo[i][0] = 3;
  TaskInfo[i][2] = i;  /* PROCESS IDENTIFIER */
  }

 for ( i = 1; i < NumProcs; i++ )
  TSKSTART( TaskInfo[i], CrayWorker, TaskInfo[i][2] );

 EnterWorker( TaskInfo[0][2] );
}

void StopWorkers()
{
 register int i;

 *SisalShutDown = TRUE;
 LeaveWorker();

 for ( i = 1; i < NumWorkers; i++ )
  TSKWAIT( TaskInfo[i] );
}

void AbortParallel()
{
 ERREXIT();
}
#endif


#ifdef SGI
```

```
static ulock_t  TheLock;
static usptr_t *UsHandle;

void ReleaseSharedMemory()
{
}

void AcquireSharedMemory( NumBytes )
int NumBytes;
{
 char ArenaName[50];

 SharedSize = NumBytes + 100000;

 sprintf( ArenaName, "/tmp/sis%d", getpid() );

 /* if ( (usconfig( CONF_INITSIZE, 1000 )) == -1 )
  SisalError( "AcquireSharedMemory", "USCONFIG CONF_INITSIZE FAILED" ); */

 if ( (usconfig( CONF_ARENATYPE, US_SHAREDONLY )) == -1 )
  SisalError( "AcquireSharedMemory", "USCONFIG CONF_ARENATYPE FAILED" );

 if ( (UsHandle = usinit(ArenaName)) == NULL)
  SisalError( "AcquireSharedMemory", "USINIT FAILED" );

 if ( (TheLock = usnewlock( UsHandle )) == (ulock_t) NULL )
  SisalError( "AcquireSharedMemory", "usnewlock FAILED" );


 SharedBase = SharedMemory = (char *) malloc( SharedSize-40 );

 if ( SharedMemory == (char *) NULL )
  SisalError( "AcquireSharedMemory", "malloc FAILED" );

 SharedMemory = ALIGN(char*,SharedMemory);
}

static void SgiTransfer( ProcId )
int ProcId;
{
 GetProcId = ProcId;

 if ( NumWorkers > 1 )
  if ( BindParallelWork )
   if ( sysmp( MP_MUSTRUN, ProcId ) == -1 )
    SisalError( "SgiTransfer", "sysmp MP_MUSTRUN FAILED" );

 EnterWorker( ProcId );

 if ( ProcId != 0 ) {
  LeaveWorker();
  _exit( 0 );
  }
}
```

```
void StartWorkers()
{
 register int NumProcs = NumWorkers;

 while( --NumProcs > 0 )
  if ( sproc( SgiTransfer, PR_SADDR, NumProcs ) == -1 )
   SisalError( "StartWorkers", "sproc FAILED" );

 if ( NumWorkers > 1 )
  if ( schedctl( SCHEDMODE, SGS_GANG, 0 ) == -1 )
   SisalError( "StartWorkers", "schedctl FAILED" );

 SgiTransfer( NumProcs );
}

void StopWorkers()
{
 *SisalShutDown = TRUE;
 LeaveWorker();
}

void AbortParallel()
{
 kill( 0, SIGKILL );
}

int MyLock(plock)
register volatile LOCK_TYPE *plock;
{
 for (;;) {
  while (*(plock) == 1);
  ussetlock(TheLock);
  if (*(plock) == 0) {
   *(plock) = 1;
   usunsetlock(TheLock);
   break;
   }
  usunsetlock(TheLock);
  }
}

int MyUnlock(plock)
register volatile LOCK_TYPE *plock;
{
 *plock = 0;
}

int MyInitLock(plock)
register volatile LOCK_TYPE *plock;
{
 *plock = 0;
}
```

```
BARRIER_TYPE *MyInitBarrier()
{
 barrier_t *bar;

 if ( (bar = new_barrier( UsHandle )) == (barrier_t *) NULL )
  SisalError( "myinitbarrier", "new_barrier FAILED" );

 init_barrier(bar);

 return( (BARRIER_TYPE*) bar );
}

int MyBarrier( bar, limit )
BARRIER_TYPE *bar;
int limit;
{
 barrier( (barrier_t *) bar, limit );
}
#endif
```

```
#include "world.h"

/* *****                                    */
/* ***** SISAL Run Time Support Software    */
/* ***** Parallel dynamic storage allocation section */
/* ***** Author:  R. R. Oldehoeft           */
/* ***** Modifier: D. C. Cann               */

/* shared LOCK_TYPE clock; */

#define NR_ZERO_BL  1
static int nr_zero_bl = NR_ZERO_BL;

struct top {
 struct top *frwd, *bkwd;
 int size;
 int lsize;
 int PId;
 char status;
 LOCK_TYPE lock;
 };

struct bot { struct top *top_ptr; };


#define  TOPSIZE   SIZE_OF(struct top)
#define  BOTSIZE   SIZE_OF(struct bot)
#define  SIZETAGS  (TOPSIZE + BOTSIZE)


struct bot *dsorg;
struct top *zero_bl;
struct top *caches;

int    *zb_start;
LOCK_TYPE *coal_lock;

struct top *btop;

static int xfthresh;
static int maxsize;

#ifdef DSA_DEBUG
int dallocs = 0;
int dfrees  = 0;
int dbytes  = 0;
#endif


void ShutDownDsa()
{
#ifdef DSA_DEBUG
fprintf( stderr, "D - (ShutDownDsa) Allocs %d Frees %d Lost Bytes %d\n",
      dallocs, dfrees, dbytes );
```

```
#endif
}


void InitDsa( size, xft )
int size;
int xft;
{
  register struct top *cu, *nx;
  register struct bot *cubot;
  register int        i, inc, roundsize;

  if ( NumWorkers == 1 ) nr_zero_bl = 1;

  /* MY_SINIT_LOCK(&clock); */
  caches  = (struct top*) SharedMalloc( sizeof(struct top) * MAX_PROCS );
  /* ALLOCATE SOME EXTRA JUST TO MAKE SURE EVERY THING IS OK!!! HELP :-) */
  zero_bl = (struct top*) SharedMalloc( sizeof(struct top) * (nr_zero_bl+5) );

  zb_start  = (int*) SharedMalloc( sizeof(int) );
  *zb_start = 0;

  coal_lock = (LOCK_TYPE*) SharedMalloc( sizeof(LOCK_TYPE) );

  for(i=0; i<NumWorkers; i++){
   caches[i].size  = 0;
   caches[i].lsize = 0;
   caches[i].frwd  = 0;
   caches[i].bkwd  = 0;
   }

  xfthresh = xft >= 0 ? xft : 0;

  /* Begin with an allocated bottom tag boundary (whose top part is */
  /* the boundary at the other end of the space!).                  */

  roundsize = ALIGN(int,size);
  dsorg = (struct bot *) SharedMalloc( roundsize );
  maxsize   = roundsize - 2 * SIZETAGS;

  /* Surround the space with an allocated bottom tag before and an
     allocated top tag after. */
  btop = (struct top *)((PCMCAST)dsorg + roundsize - TOPSIZE);
  btop->status = 'A';
  btop->size   = -1;
  btop->frwd   = btop;
  MY_SINIT_LOCK(&(btop->lock));
  btop->bkwd   = btop;

  dsorg->top_ptr = btop;

  /* Set up the space between the boundaries as free blocks, with one of  */
  /* the zero size blocks preceding each one, all in a doubly linked list */
```

```
  cu = (struct top *)((PCMCAST)dsorg + BOTSIZE);

  inc  = ((int) ((PCMCAST)btop - (PCMCAST)dsorg)) / nr_zero_bl;
  inc -= (inc % ALIGN_SIZE);

  for( i = 0; i < nr_zero_bl - 1; i++ ){
   /* Make a free block at cu */
   nx = (struct top *)((PCMCAST)cu + inc);

   cubot = (struct bot *)((PCMCAST)nx - BOTSIZE);
   cubot->top_ptr = cu;

   cu->status = 'F';
   cu->size   = (int) ((PCMCAST)cubot - (PCMCAST)cu - TOPSIZE);
   MY_SINIT_LOCK(&(cu->lock));

   /* Link it between zero size blocks */
   zero_bl[i].frwd   = cu;
   zero_bl[i].status = 'F';
   zero_bl[i].size   = 0;

   MY_SINIT_LOCK(&(zero_bl[i].lock));

   cu->bkwd = &(zero_bl[i]);
   cu->frwd = &(zero_bl[i+1]);

   zero_bl[i+1].bkwd = cu;
   cu = nx;
   }

  /* Construct last free block and link in */
  cu->frwd   = &(zero_bl[0]);
  cu->bkwd   = &(zero_bl[nr_zero_bl-1]);
  cu->status = 'F';

  cubot = (struct bot *)((PCMCAST)btop - BOTSIZE);
  cubot->top_ptr = cu;

  cu->size = (int) (( (PCMCAST)btop - (PCMCAST)cu ) - SIZETAGS);

  MY_SINIT_LOCK(&(cu->lock));
  zero_bl[nr_zero_bl-1].frwd   = cu;
  zero_bl[nr_zero_bl-1].status = 'F';
  zero_bl[nr_zero_bl-1].size   = 0;
  MY_SINIT_LOCK(&(zero_bl[nr_zero_bl-1].lock));

  zero_bl[0].bkwd = cu;

  /* Unlock the coalescing lock */
  MY_SINIT_LOCK(coal_lock);
}


/* Allocate storage from the boundary tag-managed pool */
```

```
static char *btAlloc(size)
register int size;
{
  register struct top *cu, *pr, *back;
  struct top        *newtop;
  char              *addr;
  struct bot        *cubot, *newbot;
  int               mystart, lsize;
  int               pID;

  lsize = size;
  size  = ALIGN(int,size);

  GETPROCID(pID);

  IncStorageUsed(pID,(size+SIZETAGS));
  IncStorageWanted(pID,(size));

  /* Get a free block */
  if ( NumWorkers > NR_ZERO_BL )
    *zb_start = mystart = (*zb_start + 1 + pID) % nr_zero_bl;
  else
    mystart = pID;

  pr = &(zero_bl[ mystart ]);

  back = pr;

  MY_SLOCK(&(pr->lock));

  for ( ;; ) {
    cu = pr->frwd;

    if ( cu == back ) {
      MY_SUNLOCK(&(pr->lock));
      return( 0 );
      }

    MY_SLOCK(&(cu->lock));

    if( cu->size >= size )
      break;

    MY_SUNLOCK(&(pr->lock));
    pr = cu;
    }


  if (cu->size - size <= xfthresh + SIZETAGS){
    /* Exact fit (or close enough).  Unlink cu */
    pr->frwd = cu->frwd;

    cu->frwd->bkwd = pr;
```

```
    cu->status    = 'A';
    cu->PId       = pID;
    cu->lsize     = lsize;

    addr = (char *) ((PCMCAST)cu + TOPSIZE);
   }else{
    /* Split this large block to satisfy request */
    cubot  = (struct bot *)((PCMCAST)cu + cu->size + TOPSIZE);

    addr   = (char *)((PCMCAST)cubot - size);

    newtop = (struct top *)((PCMCAST)addr - TOPSIZE);
    newbot = (struct bot *)((PCMCAST)newtop - BOTSIZE);

    /* Reduce size of cu and make new bottom tag */
    cu->size -= size + SIZETAGS;

    newbot->top_ptr = cu;

    /* Make tags for new allocated block */
    newtop->size   = size;
    newtop->status = 'A';
    newtop->PId    = pID;
    newtop->lsize  = lsize;

    MY_SINIT_LOCK(&(newtop->lock));

    cubot->top_ptr = newtop;
   }

  MY_SUNLOCK(&(pr->lock));
  MY_SUNLOCK(&(cu->lock));

  return(addr);
}


static int btDeAlloc(ptr)
register struct top *ptr;
{
  register struct top *bl_above, *bl_below, *pr, *cu;
  struct bot        *bot_above, *cubot;
  int               mystart;
  int               pID;

  GETPROCID(pID);

  bl_below  = (struct top *)((PCMCAST)ptr + ptr->size + SIZETAGS);
  bot_above = (struct bot *)((PCMCAST)ptr - BOTSIZE);

  /* Become the only coalescing process (sigh) */
  MY_SLOCK(coal_lock);

  /* Attempt to coalesce the free block below with this block */
```

```
 if( bl_below != btop ){
  for (;;) {
   MY_SLOCK(&(bl_below->lock));

   if ( bl_below->status != 'F' ) {
    MY_SUNLOCK(&(bl_below->lock));
goto tryabove;
    }

   pr = bl_below->bkwd;
   MY_SUNLOCK(&(bl_below->lock));

   MY_SLOCK(&(pr->lock));
   if( pr->status == 'F' && pr->frwd == bl_below )
break;
   MY_SUNLOCK(&(pr->lock));
   }

  MY_SLOCK(&(bl_below->lock));
  pr->frwd = bl_below->frwd;

  bl_below->frwd->bkwd = pr;

  cubot = (struct bot *)((PCMCAST)bl_below + bl_below->size + TOPSIZE);
  cubot->top_ptr = ptr;

  ptr->size += bl_below->size + SIZETAGS;

  MY_SUNLOCK(&(pr->lock));
  MY_SUNLOCK(&(bl_below->lock));
  }

tryabove:

 /* Attempt to coalesce with the block above */
 if ( bot_above != dsorg ) {
  for (;;) {
   bl_above = bot_above->top_ptr;

   MY_SLOCK(&(bl_above->lock));
   if( (PCMCAST)ptr == ((PCMCAST)bl_above + bl_above->size + SIZETAGS))
    break;
   MY_SUNLOCK(&(bl_above->lock));
   }

  if ( bl_above->status == 'F' ) {
   bl_above->size += ptr->size + SIZETAGS;

   cubot = (struct bot *)((PCMCAST)ptr + ptr->size + TOPSIZE);
   cubot->top_ptr = bl_above;

   MY_SUNLOCK(&(bl_above->lock));
   MY_SUNLOCK(coal_lock);
```

```
    /* Report new block size */
    return;
    }

  MY_SUNLOCK(&(bl_above->lock));
  }

 /* Cannot merge.  Add this block to the free list */
 MY_SUNLOCK(coal_lock);

 if ( NumWorkers > NR_ZERO_BL )
  *zb_start = mystart = (*zb_start + 1 + pID) % nr_zero_bl;
 else
  mystart = pID;

 pr = &(zero_bl[ mystart ]);
 MY_SLOCK(&(pr->lock));

 cu = pr->frwd;

 if ( pr != cu )
  MY_SLOCK(&(cu->lock));

 pr->frwd = ptr;

 ptr->bkwd   = pr;
 ptr->frwd   = cu;
 ptr->status = 'F';

 cu->bkwd = ptr;

 MY_SUNLOCK(&(pr->lock));

 if ( pr != cu )
  MY_SUNLOCK(&(cu->lock));
}

/* Return blocks from cache p to the boundary tag pool */

static int OldZap(p)
struct top *p;
{
 struct top *q;

#ifdef DSA_DEBUG
 fprintf( stderr, "D - IN OldZAP!!!\n" );
#endif

 do {
  for ( q = p->bkwd; q != 0; q = p->bkwd ) {
   p->bkwd = q->bkwd;
   btDeAlloc(q);
   }
```

```
    q = p;
    p = p->frwd;
    btDeAlloc(q);
    }
  while( p != 0 );
}


POINTER Alloc(size)
register int size;
{
  register struct top *cu, *pr, *ptr;
  register char      *addr;
  int                pID;

  GETPROCID(pID);
  pr = &caches[pID];

  /* Search for a block of exactly the right size. */
  for ( cu = pr->frwd; cu != 0; cu = cu->frwd ){
    if ( cu->lsize == size ) {
      if (cu->bkwd == 0){
        addr = (char *)((PCMCAST)cu + TOPSIZE);
#ifdef DSA_DEBUG
cu->status = 'A';
dallocs++;
/* fprintf( stderr, "ALLOCATION %x %d\n", addr, size ); */
dbytes += size;
#endif
        pr->frwd = cu->frwd;
        return( (POINTER) addr);
        }

      /* There are at least two blocks this size. */
      addr = (char *)((PCMCAST)(cu->bkwd) + TOPSIZE);
#ifdef DSA_DEBUG
cu->bkwd->status = 'A';
/* fprintf( stderr, "ALLOCATION %x %d\n", addr, size ); */
dallocs++;
dbytes += size;
#endif
      cu->bkwd = cu->bkwd->bkwd;
      return( (POINTER) addr);
      }

    pr = cu;
    }

  addr = btAlloc( size );

  if ( addr != 0 ) {
#ifdef DSA_DEBUG
/* fprintf( stderr, "ALLOCATION %x %d\n", addr, size ); */
dallocs++;
```

```
dbytes += size;
#endif
   return(  (POINTER) addr );
   }

 /* flush the cache into boundary tag system. NOTE: ALL THE CACHES, */
 /* AND NOT JUST THIS ONE, SHOULD BE FLUSHED.                       */
 DsaHelp();

 /* try boundary tag system one more time */
 addr = btAlloc( size );

 if ( addr != 0 ) {
#ifdef DSA_DEBUG
/* fprintf( stderr, "ALLOCATION %x %d\n", addr, size ); */
dallocs++;
dbytes += size;
#endif
   return(  (POINTER) addr );
   }

 SisalError( "Alloc", "ALLOCATION FAILURE: increase -ds value" );
}


/* "Shape up" the dsa system in a last ditch attempt to avoid deadlock */
/* on data memory.                                                     */

int DsaHelp()
{
 register int pID;

 GETPROCID(pID);

 IncDsaHelp(pID);

 if ( caches[pID].frwd != 0 ) {
   OldZap(caches[pID].frwd);
   caches[pID].frwd = 0;
   }
}


void DeAllocToBt( x )
POINTER x;
{
  btDeAlloc( (struct top *)((PCMCAST)x - TOPSIZE) );
}


void DeAlloc( x )
POINTER x;
{
  register struct top *pr, *cu, *ptr;
```

```
  register int size;
  register int pID;

#ifdef DSA_DEBUG
if ( x == NULL ) SisalError( "DeAlloc", "NULL POINTER ON DeAlloc!!!" );
#endif

  ptr  = (struct top *)((PCMCAST)x - TOPSIZE);

  size = ptr->lsize;

#ifdef DSA_DEBUG
if ( ptr->status != 'A' ) SisalError( "DeAlloc", "MULTIPLE DEALLOCS!!!" );
ptr->status = 'C';
/* fprintf( stderr, "FREE %x %d\n", x, size ); */
dfrees++;
dbytes -= size;
#endif

  if ( Sequential )
    pr  = &caches[ptr->PId];
  else {
    GETPROCID(pID);
    pr = &caches[ptr->PId = pID];
    }

  cu = pr->frwd;

  for ( ;; ) {
    if ( cu == 0 ) {
      pr->frwd = ptr;
      ptr->bkwd = 0;
      ptr->frwd = 0;
      return;
      }

    if ( cu->lsize == size ) {
      ptr->bkwd = cu->bkwd;
      cu->bkwd  = ptr;
      return;
      }

    pr = cu;
    cu = pr->frwd;
    }
}
```

```
#include "world.h"


#define FIBREIN         1
#define FIBREOUT        2

#define GET_Tmp(y)      (Tmp = atoi( &(argv[ ArgIndex ][(y)]) ))

#define CASE_OPTION(f,s)  if ( argv[ArgIndex][s] == (f) )


void ParseCommandLine( argc, argv )
int   argc;
char *argv[];
{
 register int   ArgIndex;
        int   Tmp;
        int   Fd;
        int   FibreFileMode;

 FibreFileMode = FIBREIN;

 for ( ArgIndex = 1; ArgIndex < argc; ArgIndex++ ) {
    if ( argv[ArgIndex][0] != '-' ) {
     switch ( FibreFileMode ) {
 case FIBREIN:
         OPEN( FibreInFd, argv[ArgIndex], "r" );
         break;

 case FIBREOUT:
         OPEN( FibreOutFd, argv[ArgIndex], "w" );
         break;

       default:
         goto Argument_Error;
         break;
       }

     FibreFileMode++;
     continue;
     }

    CASE_OPTION( 'g', 1 ) {
CASE_OPTION( 's', 2 ) {
 CASE_OPTION( 's', 3 ) {
         UseGss = TRUE;
  continue;
        }
 }

goto Argument_Error;
      }

    CASE_OPTION( 'n', 1 ) {
```

```
CASE_OPTION( 'b', 2 ) {
        BindParallelWork = FALSE;
  continue;
        }

goto Argument_Error;
      }

    CASE_OPTION( 'b', 1 ) {
      BindParallelWork = TRUE;
      continue;
}

    CASE_OPTION( 'z', 1 ) {
      NoFibreOutput = TRUE;
      continue;
      }

    CASE_OPTION( 'a', 1 ) {
CASE_OPTION( 'x', 2 ) {
 if ( GET_Tmp( 3 ) < 0 )
   goto Argument_Error;

        ArrayExpansion = Tmp;
  continue;
        }

      goto Argument_Error;
}

    CASE_OPTION( 'l', 1 ) {
      CASE_OPTION( 's', 2 ) {
        if ( GET_Tmp( 3 ) <= 0 )
          goto Argument_Error;

        LoopSlices = Tmp;
        continue;
        }

goto Argument_Error;
      }

    CASE_OPTION( 'w', 1 ) {
      if ( GET_Tmp( 2 ) <= 0 )
        goto Argument_Error;

      if ( Tmp > MAX_PROCS )
        goto Argument_Error;

      NumWorkers = Tmp;
      continue;
      }

    CASE_OPTION( 'r', 1 ) {
```

```
      GatherPerfInfo = TRUE;
      OPEN( PerfFd, "s.info", "a" );
      continue;
      }

    CASE_OPTION( 'd', 1 ) {
      CASE_OPTION( 's', 2 ) {
        if ( GET_Tmp( 3 ) <= 0 )
          goto Argument_Error;

        DsaSize = Tmp;
        continue;
        }

      CASE_OPTION( 'x', 2 ) {
        if ( GET_Tmp( 3 ) < 0 )
          goto Argument_Error;

        XftThreshold = Tmp;
        continue;
        }
}

    CASE_OPTION( '\0', 1 ) {
      FibreFileMode++;
      continue;
      }

    goto Argument_Error;
    }

 if ( LoopSlices == -1 )
   LoopSlices = NumWorkers;
 else if ( UseGss )
   SisalError( "COMMAND LINE CONFLICT", "-gss AND -ls" );

 return;

Argument_Error:
 SisalError( "ILLEGAL COMMAND LINE ARGUMENT", argv[ArgIndex] );
}


static void PrintExecutionTimes()
{
 struct WorkerInfo *InfoPtr;
 int   Worker;
 double CpuUse;
 int   NumIterations;

 fprintf( PerfFd, "  CpuTime  WallTime    CpuUse\n" );

#ifdef ALLIANT
 NumIterations = 1;
```

```
#else
  NumIterations = NumWorkers;
#endif

  for ( Worker = 0; Worker < NumIterations; Worker++ ) {
    InfoPtr = &(AllWorkerInfo[ Worker ]);

    if ( InfoPtr->WallTime != 0.0 ) {
      CpuUse  = InfoPtr->CpuTime;
      CpuUse /= InfoPtr->WallTime;
      }
    else
      CpuUse = 0.0;

    fprintf( PerfFd, " %8.4f %9.4f %8.1f%%\n",
          InfoPtr->CpuTime, InfoPtr->WallTime, CpuUse * 100.0 );
    }
}


void DumpRunTimeInfo()
{
  register struct WorkerInfo *InfoPtr;
  register int    Worker;
  register double CopyInfo, ATAttempts, ATCopies, ANoOpAttempts;
  register double RBuilds;
  register double ANoOpCopies, RNoOpAttempts, RNoOpCopies, ADataCopies;
  register int    StorageUsed, StorageWanted, DsaHelp;
  register double FlopInfo, FlopCountA, FlopCountL, FlopCountI;

  FlopInfo = FlopCountA = FlopCountL = FlopCountI = 0.0;

  CopyInfo = RBuilds = ATAttempts = ATCopies = ANoOpAttempts = 0.0;
  ANoOpCopies = RNoOpAttempts = RNoOpCopies = ADataCopies = 0.0;
  StorageUsed =  StorageWanted = DsaHelp = 0;

  for ( Worker = 0; Worker < NumWorkers; Worker++ ) {
    InfoPtr = &(AllWorkerInfo[ Worker ]);

    CopyInfo     += InfoPtr->CopyInfo;
    FlopInfo     += InfoPtr->FlopInfo;

    FlopCountA   += InfoPtr->FlopCountA;
    FlopCountL   += InfoPtr->FlopCountL;
    FlopCountI   += InfoPtr->FlopCountI;

    RBuilds      += InfoPtr->RBuilds;
    ATAttempts   += InfoPtr->ATAttempts;
    ATCopies     += InfoPtr->ATCopies;
    ANoOpAttempts += InfoPtr->ANoOpAttempts;
    ANoOpCopies  += InfoPtr->ANoOpCopies;
    RNoOpAttempts += InfoPtr->RNoOpAttempts;
    RNoOpCopies  += InfoPtr->RNoOpCopies;
    ADataCopies  += InfoPtr->ADataCopies;
```

```
  DsaHelp       += InfoPtr->DsaHelp;
  StorageUsed   += InfoPtr->StorageUsed;
  StorageWanted += InfoPtr->StorageWanted;
  }

 fprintf( PerfFd, "\n\n\n" );
 fprintf( PerfFd, "  Workers  DsaSize ExactFit DsaHelps\n" );
 fprintf( PerfFd, "%9d %8db %8db %9d\n\n",
       NumWorkers, DsaSize, XftThreshold, DsaHelp );

 if ( !UseGss ) {
  fprintf( PerfFd, "    MemW     MemU LpSliceV  ArrayEx\n" );
  fprintf( PerfFd, "%8db %8db %9d %9d\n\n",
   StorageWanted, StorageUsed, LoopSlices, ArrayExpansion );
 } else {
  fprintf( PerfFd, "    MemW     MemU  GssFact  ArrayEx\n" );
  fprintf( PerfFd, "%8db %8db %9d %9d\n\n",
   StorageWanted, StorageUsed, 1, ArrayExpansion );
  }

 PrintExecutionTimes();

 if ( CopyInfo > 0.0 ) {
  fprintf( PerfFd, "\n            AtOps          AtCopies" );
  fprintf( PerfFd, "             AcOps          AcCopies\n" );
  fprintf( PerfFd, " %18.0f  %18.0f  %18.0f  %18.0f\n\n", ATAttempts,
    ATCopies, ANoOpAttempts, ANoOpCopies         );

  fprintf( PerfFd, "            RcOps          RcCopies" );
  fprintf( PerfFd, "          CharMoves\n" );
  fprintf( PerfFd, " %18.0f  %18.0f  %18.0f\n", RNoOpAttempts,
    RNoOpCopies, ADataCopies             );

  fprintf( PerfFd, "          RBuilds\n" );
  fprintf( PerfFd, " %18.0f\n", RBuilds );
  }

 if ( FlopInfo > 0.0 ) {
  fprintf( PerfFd, "\n FlopCounts (ARITHMETIC): %18.0f\n", FlopCountA );
  fprintf( PerfFd, "              (LOGICAL): %18.0f\n", FlopCountL );
  fprintf( PerfFd, "            (INTRINSIC): %18.0f\n", FlopCountI );
  }

 fprintf( PerfFd, "\n" );
}


void InitSisalRunTime()
{
 AcquireSharedMemory( DsaSize );

 InitDsa( DsaSize, XftThreshold );
```

```
  InitErrorSystem();
  InitWorkers();
  InitReadyList();
  InitSignalSystem();
  InitSpawn();
}
```

```
#include "world.h"


static void ConfigureExecution( lsValue, gssValue, bValue, xftValue, axValue )
int lsValue;
int gssValue;
int bValue;
int xftValue;
int axValue;
{
 if ( lsValue > 0 ) {
  if ( gssValue == 1 )
   SisalError( "sconfig", "-gss AND -ls CONFLICT" );

  LoopSlices = lsValue;
  }

 if ( gssValue == 1 ) {
  if ( LoopSlices > 0 )
   SisalError( "sconfig", "-gss AND -ls CONFLICT" );

  UseGss = TRUE;
  }

 if ( bValue == 1 )
  BindParallelWork = TRUE;

 if ( axValue >= 0 )
  ArrayExpansion = axValue;

 if ( xftValue >= 0 )
  XftThreshold = xftValue;
}

static void ParseInterfaceArguments( wValue, dsValue, rValue )
int wValue;
int dsValue;
int rValue;
{
 if ( dsValue > 0 )
  DsaSize = dsValue;
 else
  SisalError( "sstart", "ILLEGAL -ds VALUE" );

 if ( wValue > 0 && wValue <= MAX_PROCS )
  NumWorkers = wValue;
 else
  SisalError( "sstart", "ILLEGAL -w VALUE" );

 if ( rValue == TRUE ) {
  GatherPerfInfo = TRUE;
  OPEN( PerfFd, "s.info", "a" );
  }
```

```
 if ( LoopSlices == -1 )
   LoopSlices = NumWorkers;
}


#define SCONFIG_FUNCTION(x)                                    \
void x( lsValue, gssValue, bValue, xftValue, axValue )          \
int *lsValue;                                                   \
int *gssValue;                                                  \
int *bValue;                                                    \
int *xftValue;                                                  \
int *axValue;                                                   \
{                                                              \
 ConfigureExecution( *lsValue, *gssValue, *bValue, *xftValue, *axValue ); \
}

#define SSTART_FUNCTION(x)                                      \
void x( wValue, dsValue, rValue )                              \
int *wValue;                                                   \
int *dsValue;                                                  \
int *rValue;                                                   \
{                                                              \
 ParseInterfaceArguments( *wValue, *dsValue, *rValue ); \
 InitSisalRunTime();                                           \
 StartWorkers();                                               \
}

#define SSTOP_FUNCTION(x) \
void x()                \
{                       \
 StopWorkers();         \
 ShutDownDsa();         \
 ReleaseSharedMemory(); \
 if ( GatherPerfInfo )  \
   DumpRunTimeInfo();   \
}


/* C VERSIONS */
SSTART_FUNCTION( sstart )
SSTOP_FUNCTION( sstop )
SCONFIG_FUNCTION( sconfig )

/* FORTRAN VERSIONS: CRAY, OTHERS */
SSTART_FUNCTION( SSTART )
SSTOP_FUNCTION( SSTOP )
SSTART_FUNCTION( sstart_ )
SSTOP_FUNCTION( sstop_ )
SCONFIG_FUNCTION( SCONFIG )
SCONFIG_FUNCTION( sconfig_ )


#define IDInfo( x, y, z, w ) \
{ \
```

```
 register int  i; \
 register int  DimInc; \
 register int  InfoInc; \
 register int  DSize  = 1; \
 register int  Major  = z[0]; \
 register int  Mode   = z[1]; \
 register int  Mutable = w; \
 z += 3; \
 switch ( Major ) { \
  case ROW_MAJOR: \
   InfoInc = -5; \
   z  += ((x-1)*5); \
   if ( Mode == PRESERVE ) { \
    DimInc = -1; \
    y += (x-1); \
    } \
   else \
    DimInc = 1; \
   break; \
  case COL_MAJOR: \
   InfoInc = 5; \
   if ( Mode == PRESERVE ) \
DimInc = 1; \
   else { \
DimInc = -1; \
    y += (x-1); \
} \
   break; \
  default: \
   SisalError( "Mixed Language Interface", "ILLEGAL ARRAY DESCRIPTOR" ); \
  } \
 if ( DimInc == 1 && x != 1 ) \
  Mutable = FALSE; \
 for ( i = 0; i < x; i++ ) { \
  y->LSize  = z[LHI]-z[LLO]+1; \
  y->Offset  = z[LLO]-z[PLO]; \
  y->SLow   = z[SLO]; \
  y->DSize  = DSize; \
  y->Mutable = Mutable; \
  DSize *= (z[PHI]-z[PLO]+1); \
  y += DimInc; \
  z  += InfoInc; \
  } \
}


void InitDimInfo( ronly, Dim, DimInfo, Info )
int     ronly;
int     Dim;
DIMINFOP  DimInfo;
int    *Info;
{
#if SUNIX || ALLIANT || CRAY || SUN || SGI || RS6000
 IDInfo( Dim, DimInfo, Info, Info[2] || ronly );
```

```
#else
  IDInfo( Dim, DimInfo, Info, FALSE );
#endif
}

void IDescriptorCheck( Dim, Info )
int  Dim;
int *Info;
{
 register int CurrentDim;
 register int Plo,Phi, Llo,Lhi;
 register int Major;

 Plo = Phi = Llo = Lhi = CurrentDim = 0;

 switch ( (Major = *Info) ) {
  case ROW_MAJOR:
  case COL_MAJOR:
   break;

  default:
   goto InfoError;
  }

 Info = &(Info[3]);

 for ( CurrentDim = 1; CurrentDim <= Dim; CurrentDim++ ) {
  Plo = Info[PLO];
  Phi = Info[PHI];
  Llo = Info[LLO];
  Lhi = Info[LHI];

  if ( Phi-Plo+1 == 0 ) {
   if ( Lhi-Llo+1 != 0 ) goto InfoError;
   continue;
   }

  if ( Lhi-Llo+1 == 0 ) {
   if ( Phi-Plo+1 < 0 ) goto InfoError;
   continue;
   }

  if ( Lhi-Llo+1 < 0 ) goto InfoError;
  if ( Phi-Plo+1 < 0 ) goto InfoError;

  if ( Llo < Plo || Llo > Phi ) goto InfoError;
  if ( Lhi < Plo || Lhi > Phi ) goto InfoError;

  Info = Info+5;
  }

 return;

InfoError:
```

```
 if ( UsingSdbx )
  SdbxMonitor( SDBX_IERR );

 fprintf( stderr,
  "Descriptor Info: Major=%d Dimensions=%d Current Dimmension = %d\n",
  Major, Dim, CurrentDim );

 fprintf( stderr, "Descriptor Info: Plo=%d Phi=%d Llo=%d Lhi=%d\n",
  Plo,Phi,Llo,Lhi );

 SisalError( "IDescriptorCheck", "ILLEGAL INTERFACE ARRAY DESCRIPTOR" );
}
```

```
#include "world.h"


main( argc, argv )
int    argc;
char **argv;
{
  ParseCommandLine( argc, argv );
  InitSisalRunTime();

  fprintf( stderr, "%s %s\n", BANNER, VERSION );

  SisalMainArgs = ReadFibreInputs();

  StartWorkers();
  SisalMain( SisalMainArgs );
  StopWorkers();

  if ( !NoFibreOutput )
    WriteFibreOutputs( SisalMainArgs );

  if ( GatherPerfInfo )
    DumpRunTimeInfo();

  ShutDownDsa();

  exit(0);
}
```

LIST OF REFERENCES

1. Chambers, F.B., Duce, D.A., and Jones, G.A., eds., *Distributed Computing*, Academic Press, 1984.

2. Cann, D.C., "SISAL 1.2:A Brief Introduction and Tutorial," documentation included in the downloaded SISAL project files by anonymous ftp from Lawrence Livermore National Laboratories.

3. Deitel, H.M., *Operating Systems Second Edition*, pp. 313-357, Addison-Wesley, 1990.

4. Cann, D.C., et al, "SISAL Reference Manual, Language version 2.0," documentation included in the downloaded SISAL project files by anonymous ftp from Lawrence Livermore National Laboratories.

5. Cann, D.C., et al, "The Optimizing SISAL Compiler: Version 12.0," documentation included in the downloaded SISAL project files by anonymous ftp from Lawrence Livermore National Laboratories.

INITIAL DISTRIBUTION LIST

Defense Technical Information Center Cameron Station Alexandria, VA 22304-6145	2
Library, Code 52 Naval Postgraduate School Monterey, California 93943-5002	2
Chairman, Code EC Department of Electrical and Computer Engineering Naval Postgraduate School Monterey, California 93943-5000	1
Prof. D. J. Fouts, Code EC/Fs Department of Electrical and Computer Engineering Naval Postgraduate School Monterey, California 93943-5000	2
Prof. S. B. Shukla, Code EC/Sh Department of Electrical and Computer Engineering Naval Postgraduate School Monterey, California 93943-5000	1
LT Stanely L. Fox II Puget Sound Naval Shipyard, Code 811 Bremerton, Washington 98314-5000	1